# VERIGRAPHI: A MULTI-AGENT FRAMEWORK OF HIERARCHICAL RTL GENERATION FOR LARGE HARDWARE DESIGNS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Generating synthesizable Verilog for large, hierarchical hardware designs remains a significant challenge for large language models (LLMs), which struggle to replicate the structured reasoning that human experts employ when translating complex specifications into RTL. When tasked with producing hierarchical Verilog, LLMs frequently lose context across modules, hallucinate interfaces, fabricate inter-module wiring, and fail to maintain structural coherence—failures that intensify as design complexity grows and specifications involve informal prose, figures, and tables that resist direct operationalization. To address these challenges, we present VeriGraphi, a framework that introduces a spec-anchored Knowledge Graph as the central architectural substrate driving the RTL generation pipeline. VeriGraphi constructs a Hierarchical Design Architecture (HDA), a structured knowledge graph that explicitly encodes module hierarchy, port-level interfaces, wiring semantics, and inter-module dependencies as first-class graph entities and relations. Built through iterative multi-agent analysis of the original specification, this Knowledge Graph provides a deterministic, machine-checkable structural scaffold before code generation. Guided by the KG, a progressive coding module incrementally generates pseudo-code and synthesizable RTL while enforcing interface consistency and dependency correctness at each submodule stage. We evaluate VeriGraphi on a benchmark comprising three representative specification documents published by the National Institute of Standards and Technology (NIST) and their corresponding implementations, and we present a RISC-V 32I processor as a detailed case study to illustrate the full pipeline. The results demonstrate that VeriGraphi enables reliable hierarchical RTL generation with minimal human intervention for RISC-V, marking a meaningful progress for LLM-generated hardware design while maintaining strong functional correctness.

## 1 INTRODUCTION

The rapid growth of large language models (LLMs) has significantly accelerated the adoption of artificial intelligence (AI) across diverse application domains, leading to an increasing imbalance between the pace of AI software innovation and the rate of underlying hardware advancement. While hardware remains the fundamental enabler of scalable and energy-efficient AI systems, progress in hardware design automation continues to lag due to the intrinsic complexity of hardware architectures and the heavy reliance on human expertise. As a result, scalable chip design has emerged as a critical bottleneck in supporting the momentum of AI-driven innovation.

State-of-the-art LLMs can now automate substantial portions of software development, generating syntactically correct and semantically meaningful code in languages such as Python and C++, even for modular, object-oriented designs (Hou et al., 2024; Rahman et al., 2025). Hardware description language (HDL) generation, particularly Verilog, remains far more challenging: hardware designs are hierarchical, structurally constrained, and timing-sensitive, requiring precise coordination among modules, interfaces, and signals. LLMs struggle to generate complex hierarchical Verilog that is both syntactically correct and functionally faithful, especially for rarely implemented or specialized designs (Tang et al., 2025; Yang et al., 2025a; Yu et al., 2025), so human intervention is often needed to fix hallucinations, misaligned hierarchies, and incorrect wiring.



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

A fundamental limitation arises from the gap between the unstructured nature of real-world hardware specification documents and the requirements of HDL generation. Specs are typically lengthy documents that contain informal text, tables, figures, and equations, making it difficult for LLMs to directly extract and operationalize design intent (Yu et al., 2025). Flat prompting further exacerbates this issue, frequently causing LLMs to lose context, invent missing structural details, or incorrectly infer module connectivity when tasked with generating Verilog directly from specifications.

To address these challenges, recent work has explored hierarchical prompting and agent-based multi-stage pipelines that decompose specifications into submodules and iteratively refine RTL (Nakkab et al., 2024; Tang et al., 2025; Yu et al., 2025), as well as graph-based representations that model architectural entities and relationships for assertion generation, architecture-to-RTL translation, and neighborhood-aware retrieval (Bai et al., 2025; Yang et al., 2025b; Jayanth et al., 2026). Retrieval- and summarization-oriented tools like NotebookLM effectively organize information from specs, but target document understanding rather than synthesis-oriented, agentic RTL generation (Google, 2024). Despite this progress, tightly integrated structural representations that ensure deterministic hierarchy construction, interface consistency, and end-to-end traceability remain lacking.

Motivated by these limitations, we propose a knowledge-graph–driven pipeline that unifies specification analysis and RTL generation. Rather than relying on text-centric plans or post-hoc structural recovery, it constructs a spec-anchored knowledge graph that explicitly captures hierarchy, interfaces, and inter-module relationships before code generation. This structure decomposes a complex RTL task into module-level problems that LLMs can more easily generate and revise, enables bottom-up verification for earlier bug detection and clearer fault localization, and supports deterministic integration, consistent interfaces, and cross-stage traceability by reducing implicit structural inference. The knowledge graph thus serves as an LLM-friendly architectural plan throughout the pipeline.

The main contributions of this work are summarized below:

- We propose a spec-anchored **Knowledge Graph (KG)**–driven RTL generation framework integrating architectural analysis, hierarchical modeling, and progressive coding within a unified agentic pipeline, where the KG is an intermediate representation encoding modules, ports, signals, and hierarchical relations through explicit semantic edges, enabling module-level validation and deterministic structural integration prior to RTL generation.
- We design a KG-guided progressive coding mechanism that incrementally generates pseudo-code and synthesizable RTL while enforcing dependency correctness, and structural alignment with the graph model.
- We evaluate **VeriGraphi** through both synthesizability and functional correctness, using specification-driven testbenches for module-level and integration-level validation. Our results show successful generation of complex designs, including RV32I with minimal human intervention and HMAC with zero intervention, while achieving significantly fewer coding iterations, and thus lower generation cost, than prior work.

## 2 Related work

**LLM-Driven RTL Generation Pipelines.** ROME (Nakkab et al., 2024) introduced hierarchical prompting and one of the earliest LLM-generated processors, but relies on expert-provided decompositions (HDHP) or is fragile for rare designs (PGHP), leaving hierarchical design to the LLM to decide. HiVeGen (Tang et al., 2025) adds module reuse and runtime structural correction but depends on predefined templates and LLVM-based inputs from C/C++ kernels rather than arbitrary PDF specifications. ChatCPU (Wang et al., 2024) combines fine-tuned LLMs with a processor description language but is tied to predefined CPU templates. Spec2RTL-Agent (Yu et al., 2025) automates spec analysis, coding, and debugging with multiple agents, yet its text-centric intermediate plan is not machine-checkable, hindering structural verification and cross-stage error attribution. These pipelines thus still lack deterministic hierarchy construction, explicit interface modeling, and reliable top-level integration.

**Graph-Based & Retrieval-Augmented Hardware Reasoning.** AssertionForge (Bai et al., 2025) builds a knowledge graph from natural-language specifications refined with existing RTL to improve assertion generation, but assumes RTL is already available and targets verification rather than spec-first RTL generation. VeriRAG (Jayanth et al., 2026) combines hardware knowledge graphs

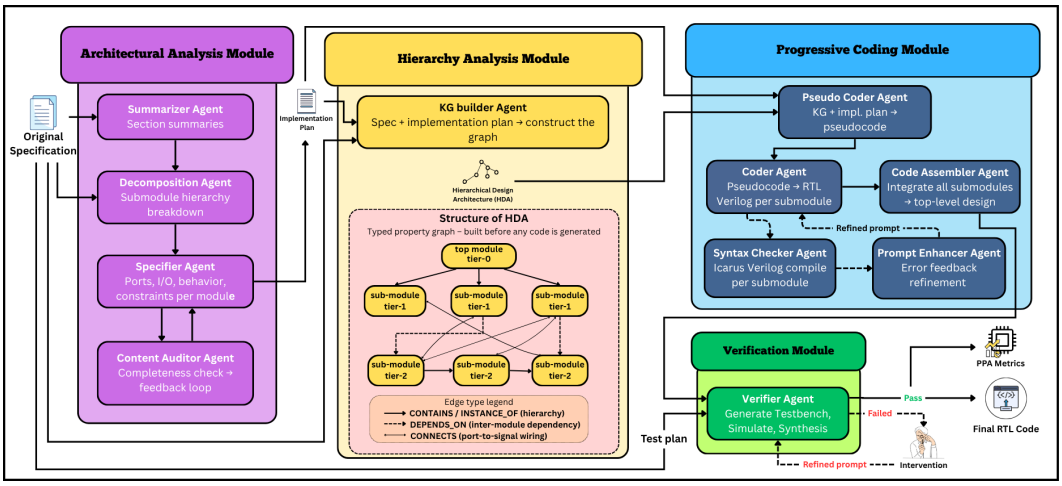


Figure 1: System Architecture.

with vector retrieval to supply structured context for Verilog and SVA generation, but its graph is built from an RTL corpus and depends on retrieval coverage, weakening traceability to specification intent and failing on rare or novel modules. Yang et al. (2025b) translate architectural descriptions into symbolic specifications via a neural-symbolic blueprint graph, but capture mainly high-level structure and behavior, with limited modeling of the port-level connectivity and wiring constraints needed for deterministic RTL integration.

# 3 Framework

## 3.1 Overview

To achieve an automated hardware design pipeline directly from raw specs, we propose a structured architecture, illustrated in Fig. 1, composed of three tightly coupled modules. Each module incrementally transforms specification-level intent into a synthesizable Verilog design, while maintaining an organized Knowledge Graph (KG) that serves as an explicit architectural representation. The KG is constructed based on the decomposed architecture and functional implementation plan to enable systematic reasoning, traceability, and deterministic module integration. The framework includes:

**Architectural Analysis Module.** This module performs an organized interpretation of the specification document, emulating the reasoning process of a hardware expert. It analyzes and summarizes the document, decomposes the design into functional submodules, and formalizes their roles, interfaces, and dependencies. The output is a detailed implementation plan that defines the architectural blueprint for later stages.

**Hierarchy Analysis Module.** Building upon the implementation plan, this module constructs and updates the Knowledge Graph. It captures the Device Under Test (DUT) interface (including ports, clock/reset conventions, parameters, and constraints) and models explicit hierarchical relationships among submodules. Wiring semantics, interface mappings, and structural dependencies are represented as typed graph entities and relations, ensuring deterministic hierarchy formation and machine-checkable connectivity.

**Progressive Coding Module.** Guided by the implementation plan and the Knowledge Graph, this module incrementally generates synthesizable Verilog code in a progressive manner. Submodules are implemented, verified, and assembled into the top-level design through an iterative closed-loop process involving code generation, syntax verification, and prompt optimization.

**Verification Module.** This module verifies the generated RTL in a staged manner, starting with submodule-level checking and ending with full top-module validation. It derives spec-grounded testbenches, runs simulation for both module-level and integration-level correctness, and finally

synthesis and PPA evaluation; only designs that pass both functional verification and synthesizability checks are accepted as final outputs, while failures trigger human intervention.

### 3.2 Architectural Analysis Module

The Architectural Analysis Module is inspired by the Iterative Understanding and Reasoning paradigm introduced in Spec2RTL (Yu et al., 2025), where the objective is to transform lengthy, unstructured, and LLM-unfriendly specification documents into a structured implementation plan. Real-world hardware specifications often contain extensive textual definition interleaved with figures, tables, and informal explanations, making direct RTL generation unreliable. This module addresses that challenge by iteratively organizing, extracting, and validating essential architectural elements to produce a well-structured, LLM-friendly implementation plan. The module is implemented as a tightly coupled multi-agent pipeline, where specialized LLM agents collaborate to progressively refine architectural intent.

**Summarizer Agent.** Processing large PDF specifications is challenging because figures and tables are often lost in raw text extraction. To address this, we use a custom preprocessing pipeline that extracts both text and visual artifacts from the specification. Text is extracted with PyPDF, while a Python-based preprocessing tool captures figures and tables as indexed image files *(e.g., figure_1, figure_2)*, preserving their original numbering and correspondence in the document. The specification PDF and these extracted visual artifacts together form the primary system input. Since hardware specifications are typically organized into sections, the *Summarizer Agent* generates structured summaries at section granularity. During summarization, it reasons over the text to detect implicit or explicit figure and table references, retrieves the corresponding pre-extracted images, and incorporates them into the reasoning process. This multimodal grounding helps capture structural and architectural details conveyed through diagrams. The output is a structured JSON summary for each section, including metadata such as referenced figures and tables, which serves as an LLM-friendly multimodal context for downstream decomposition and architectural analysis.

**Decomposition Agent.** Complex hardware systems inherently follow hierarchical design principles. The Decomposition Agent takes both the section-wise summaries and the original specification content as input to systematically identify all required sub-functions or submodules necessary to implement the target top-level design. Beyond identifying components, the agent determines a correct implementation order by explicitly modeling inter-module dependencies, producing an ordered and dependency-aware decomposition. When the specification does not explicitly define a hierarchy, the LLM is guided to infer a plausible hierarchical structure based on general architectural knowledge, which is then exposed for optional human review and refinement. A key aspect of this agent is its ability to associate each submodule with precise *spec_reference* links to relevant sections of the original document. This allows subsequent agents to operate on focused, context-specific excerpts rather than the full specification, significantly reducing token usage and improving efficiency. The output is a hierarchical structural blueprint consisting of submodules, their functional roles, dependencies, and targeted specification references, aligned with the original design intent.

**Specifier Agent.** Given the identified submodules and their associated references, the *Specifier Agent* extracts detailed implementation-relevant information for each component. Rather than reprocessing the entire spec, it operates using the structured summaries while cross-checking with the source document to maintain fidelity. For each submodule, it formalizes inputs, outputs, functional behavior, constraints, and relevant spec anchors. The agent operates in an iterative refinement loop, updating its descriptions based on feedback from the *Content Auditor Agent* to ensure completeness and consistency. The output is a structured description suitable for downstream graph construction.

**Content Auditor Agent.** The *Content Auditor Agent* reviews the generated descriptions to ensure completeness, consistency, and alignment with the specification. It evaluates each submodule description across four key dimensions: structural completeness *(e.g., presence of all required fields such as inputs/outputs with data types and bit widths)*, correctness with respect to the specification, clarity of implementation steps, and readiness for RTL coding. For instance, if a description omits signal widths, provides ambiguous step ordering, or references non-specific sections, the agent flags these as issues and suggests concrete corrections. Based on this evaluation, the agent produces structured feedback indicating whether the description is valid, along with identified issues and actionable suggestions. Descriptions that fail validation are routed back to the *Specifier Agent* for refinement,

while valid ones may still receive improvement suggestions. This feedback-driven loop ensures that each submodule description is sufficiently precise, unambiguous, and implementation-ready before proceeding to downstream structural modeling.

After completing these stages, the system produces a comprehensive and hierarchically organized implementation plan for the top module and its submodules. This structured plan serves as the foundation for the subsequent Hierarchy Analysis Module, where explicit Knowledge Graph representations are constructed from both the original specification and the refined implementation plan.

### 3.3 HIERARCHY ANALYSIS MODULE

A major limitation in prior LLM-based hardware generation is the lack of an explicit and specifiable representation of connections between submodules when composing a top-level design. In our experiments, we observed that accurate, detailed connectivity becomes crucial as designs grow in complexity and depend on multi-layer hierarchical child modules. When such detail is missing, LLMs frequently hallucinate I/O ports, internal signals, and wiring, leading to integration failures even when individual submodules appear correct.

To address this, our framework introduces a *KG Builder Agent* within the Hierarchy Analysis Module. It identifies the required modules, captures their dependencies, organizes their hierarchical relationships, and structural information needed for downstream code generation and assembly.

#### 3.3.1 **HIERARCHICAL DESIGN ARCHITECTURE (HDA)**

The *Hierarchical Design Architecture (HDA)* is a design-session knowledge graph constructed by the *KG Builder Agent* to represent the target design as an explicit, machine-readable hierarchy with well-defined wiring semantics. HDA is generated using two inputs: (1) the structured implementation plan produced by the Architectural Analysis Module, and (2) the original spec including section and figure references. The graph is implemented as a property graph consisting of nodes and typed relationships with associated key–value properties.

**Construction Workflow (Spec-Grounded)** For each module required in the target design, the *KG Builder Agent* follows a specification-grounded construction process.

**Graph construction from implementation plan.** The agent builds the HDA using the structured implementation plan together with the original specification. Based on the decomposed modules and their dependencies, it creates module nodes and establishes hierarchical relationships.

**Explicit connectivity and constraint modeling.** Ports, signals, parameters, and wiring constraints are explicitly defined using information grounded in the specification and implementation plan, rather than being inferred during code generation. This ensures consistent and unambiguous inter-module connections. The resulting HDA forms a complete and deterministic architectural scaffold capturing both hierarchy and connectivity, which guides downstream RTL generation and reduces hallucinated interfaces or incorrect wiring.

**Graph Semantics.** HDA models the target design at module granularity while making both hierarchy and connectivity explicit. Each node represents a module in the architecture, annotated with functional descriptions, interface definitions (ports and widths), configuration parameters, and optional references to specification sections or figures. Edges encode three structural relationship types: **CONTAINS / INSTANCE_OF**, hierarchical relationships describing which module calls which submodule; **DEPENDS_ON**, defining the inter-module dependencies; and **CONNECTS**, wiring relationships describing port-to-signal connections across modules.

By explicitly encoding both hierarchy and connectivity, the HDA provides a blueprint that constrains LLM-driven RTL generation and reduces the likelihood of hallucinated interfaces or wiring errors during module integration. The constructed HDA is also saved as a JSON file, together with reference links to the original specification sections, and visualized graphically to enable optional human inspection without tracing hierarchy and connectivity manually from the full document.

### 3.4 PROGRESSIVE CODING MODULE

With the implementation plan from the Architectural Analysis Module and the knowledge graph from the Hierarchy Analysis Module, the pipeline proceeds to the code generation stage, referred to as the **Progressive Coding Module**. This module consists of 5 coordinated agents that collaboratively generate compilable, hierarchical RTL in a staged, bottom-up manner guided by the HDA.

**Pseudo Coder Agent.** Rather than directly generating RTL, the pipeline first produces structured hardware pseudocode. This agent takes as input both the KG node (capturing ports, parameters, and interface constraints) and the corresponding implementation plan entry (capturing functionality and step-wise logic). It synthesizes these into pseudocode that explicitly defines control flow, data operations, and module behavior. This intermediate representation is critical for bridging high-level specification and RTL, allowing structured reasoning without syntax constraints, and is stored for subsequent use by the *Coder Agent*.

**Coder Agent.** The Coder Agent translates the generated pseudocode into clean, synthesizable Verilog code. The agent is explicitly guided to produce synthesizer-friendly RTL compatible with standard toolchains (OpenROAD Project, 2025), including compilation using *iverilog (-g2012 -t null -Wall)* and downstream synthesis. This ensures that generated code adheres to practical hardware design constraints beyond syntactic correctness.

**Syntax Checker Agent.** Each generated submodule is validated using Icarus Verilog (Williams, 2026). Early-stage compilation at the submodule level ensures that errors are detected before integration, preventing propagation across hierarchical dependencies and simplifying debugging.

**Prompt Enhancer Agent.** If compilation fails, an iterative correction loop is triggered. The *Prompt Enhancer Agent* refines the generation prompt using the latest full code, compiler error messages, and a compressed history of previous failures. To maintain efficiency, only erroneous lines and corresponding errors from prior iterations are retained. This significantly reduces token usage (by approximately 80%) while preserving essential debugging context. The refined prompt is then used to regenerate the RTL via the Coder Agent. This loop continues until successful compilation or a retry limit is reached.

**Code Assembler Agent.** Once all submodules are generated, the *Code Assembler Agent* builds the top-level hierarchical wrapper, using the KG's INSTANCE_OF and CONNECTS relations to determine module instantiations and exact port wiring, and validates it through syntax checking with limited error-feedback retries to yield a fully integrated, compilable RTL design. Overall, modules are generated bottom-up in hierarchy order (leaf modules first), so dependencies are resolved before integration; this staged approach enforces structural correctness, improves convergence efficiency, and enables reliable generation of complex hierarchical designs (Algorithm 1).

### 3.5 VERIFICATION MODULE

#### 3.5.1 VERIFIER AGENT

The Verification Module is driven by a specification-based Verifier Agent. To verify the RTL generated by VeriGraphi, the *Verification Agent* employs Claude Sonnet 4.6 (Anthropic, 2026) to construct a specification-driven verification pipeline in four sequential steps, grounded in the specification documents rather than the generated RTL, ensuring testbenches serve as independent ground truth verified against the spec.

**Step 1 — Specification Summarization** The Verification Agent is provided the full specification PDFs as project-level context. For each design, it extracts a structured per-module summary covering functional role, port-level interface, behavioral constraints, and normative requirements, following the schema shown in Listing 4. This summary is constructed without reference to the RTL, preventing test derivation from inheriting implementation-level bugs or assumptions.

**Step 2 — Module-Level Test Case Derivation** Each normative behavioral claim is mapped to a concrete, checkable condition. Boundary conditions, all-zero and all-one patterns, spec-defined constants, and known-answer pairs are extracted directly from the spec. For modules with undefined

RTL functions, tests are authored against spec requirements and annotated as expected failures with documented bug references. Representative cases are shown in Listing 5.

**Step 3 — Integration-Level Test Construction** Integration testbenches validate end-to-end data propagation through the full module hierarchy, focusing on pipeline depth timing, cross-module consistency, and X-propagation detection from broken submodules. Representative cases are shown in Listing 6. This layer provides the simulation-based pass/fail signal determining whether the assembled top-level design requires further iteration.

**Step 4 — Testbench Assembly and Coverage Verification** All submodules and integration testbenches are consolidated into a single unified file with independent top-level test modules per design, each producing parseable PASS/FAIL output consumed by an automated shell script for compilation, simulation, and aggregation. Coverage is verified by mapping every normative spec clause to at least one test case—a spec-clause standard stronger than code-path coverage; an example mapping is shown in Table 3.

**Step 5 - Synthesis and PPA Evaluation** Beyond simulation, the Verification Agent also performs synthesis-based validation. Yosys (Wolf, 2013) is used as the primary synthesis check, confirming that the generated RTL is synthesizable and mapping it to a gate-level netlist. OpenLane (Various, 2024) then runs the full physical design flow, performing broad design rule checks and reporting PPA—covering cell area, critical path delay, and power consumption—that quantify the quality of the generated RTL against standard physical implementation targets.

# 4 Case Study: RISC-V 32I Processor

## 4.1 Case Study: Agent Pipeline for RV32I Spec Processing

To demonstrate the operation of the proposed pipeline, we conducted a case study using the RV32I specification. The specification document was obtained from the official RISC-V user-level ISA documentationWaterman & Asanović (2019). Since hardware specifications are typically distributed as long PDF containing both textual descriptions and figures, the first step involves extracting structured information suitable for LLM processing.

The specification text was extracted using PyPDF, while figures and tables referenced in the document were automatically captured using a custom preprocessing tool. These visual artifacts are important because architectural specifications often encode critical information through diagrams, dataflow representations, and lookup tables. The extracted content is then processed by a set of coordinated LLM agents within the Architectural Analysis Module. We have used AutogenWu et al. (2024) framework for our multi-agent pipeline.

Following preprocessing, the extracted specification content is processed by the multi-agent pipeline implemented in the Architectural Analysis Module. All agents in this stage were implemented using the *GPT-4o-mini*OpenAI (2024) model to balance reasoning capability and computational efficiency during iterative analysis of large specification documents.

The first stage of the pipeline performs section-level summarization of the specification. Each section of the document is analyzed and converted into a structured JSON summary containing the section purpose, key technical concepts, important terminology, and cross-section dependencies. In total, the pipeline processed **88 sections** inside **238 pages** of the RISC-V specification, producing structured summaries that provide an LLM-friendly representation of the architecture while preserving implementation-relevant details. These summaries serve as the contextual foundation for downstream architectural decomposition. We filter out irrelevant sections from the summaries for the decomposition to save input tokens.

Using both the summaries and the original specification content, the decomposition and description stages generate a structured **implementation plan** representing the RV32I architecture as a collection of modular hardware components. During this stage, the content auditor agent evaluates the generated module descriptions to ensure completeness, correctness, and readiness for RTL implementation. The final output of this process is a structured JSON representation describing each

module, its interface, functionality, and implementation constraints. An excerpt of the generated implementation plan is shown on Listing 1.

This structured implementation plan is then used by the *KG Builder Agent* to construct the **Hierarchical Design Architecture (HDA)**, a machine-readable knowledge graph whose nodes represent hardware modules and whose edges capture hierarchical and wiring relationships. For RV32I, these nodes include modules such as `RISC_V_32I`, `RegisterFile`, `InstructionDecoder`, `ImmediateGenerator`, `ALU`, `ControlUnit`, `MemoryInterface`, `HintInstructionHandler`, `ExtensionManager`, and `VLIWSupport`. Fig. 2 shows the resulting RV32I graph, illustrating how major components are connected through explicit structural relations. The generated HDA is stored in JSON format and visualized through an interactive static website.This graph representation serves as the structural backbone of the pipeline. Our pipeline extracted `ExtensionManager` and `VLIWSupport` even if they are not a part of the RV32I, because the spec includes, "Supporting VLIW encodings," which explicitly describes VLIW as a RISC-V extension approach, so it's spec guided.

Given the structured implementation plan and the Hierarchical Design Architecture (HDA), the Progressive Coding Module generates RTL in a staged manner. For the RV32I case study, the Pseudo Coder Agent first produces pseudo-code for each submodule. An example pseudo-code for the ALU is shown on Listing 2.

*The Coder Agent* then translates the pseudo-code into synthesizable RTL. It is guided to generate clean, synthesizer-friendly Verilog code that adheres to standard coding practices and avoids tool-specific incompatibilities. The generated RTL is intended to be compatible with downstream compilation using Icarus Verilog and synthesis flows such as YosysWolf (2013) and OpenLaneVarious (2024) for PPA evaluation.

**Iterative Refinement.** Each generated submodule is validated using the *Syntax Checker Agent*. If compilation fails, the *Prompt Enhancer Agent* refines the generation using error feedback. As described in Section 3.4), only the latest full code and error logs are retained, while earlier iterations are summarized to reduce token usage.

**Cost Analysis.** By analyzing the cost, we have found that it requires around 0.7 USD (Listing 3).

**Functional Correctness.** We evaluate functional correctness using specification-driven testbenches applied to the RV32I design. The generated processor successfully passes all 183 test cases, including both submodule-level validation and randomized arithmetic tests for the ALU, demonstrating correct behavioral execution across key components.

### 4.2 Case Study: Other experiments

We also leverage three widely recognized cryptographic standards from the NIST Federal Information Processing Standards (FIPS): the Advanced Encryption Standard (AES)National Institute of Standards and Technology (NIST) (2023a), the Digital Signature Standard (DSS)National Institute of Standards and Technology (NIST) (2023b), and the Keyed-Hash Message Authentication Code (HMAC)National Institute of Standards and Technology (NIST) (2008). By utilizing these diverse benchmarks, we provide an evaluation of our tool across a variety of complex hardware design tasks. Their specification documents vary in size and structure, spanning 52 pages with 5 sections, 15 figures, and 8 tables for AES, 86 pages with 7 sections, 4 figures, and 4 tables for DSS, and 13 pages with 6 sections, 1 figure, and 1 table for HMAC.

These results demonstrate that the staged generation approach not only produces structurally consistent RTL but also maintains low computational cost through efficient iterative refinement. The code will be open on GitHub upon publication.

## 5 Experimental Results

Evaluation Metrics. We evaluate pipeline effectiveness using two metrics: (1) **Intervention**, defined as the total number of human fixes required across both simulation and synthesis to achieve syntactically correct RTL — counted per module and reported as the highest fix count observed across

all modules in the design; and (2) **Coding**, defined as the average number of code generation and revision iterations required per submodule, capturing the efficiency and convergence behavior of the *Progressive Coding Module*. Note that Spec2RTL-AgentYu et al. (2025) does not report intervention at per-module granularity; for a fair comparison we align our reported intervention figure to their definition by taking the per-module maximum.

Table 1: PERFORMANCE COMPARISON: VERIGRAPHI VS. SPEC2RTLYU ET AL. (2025) (LOWER IS BETTER).

| | RISC-V-32I | | DSS | | HMAC | | AES | |
|---|---|---|---|---|---|---|---|---|
| Metric | VG | S2R | VG | S2R | VG | S2R | VG | S2R |
| Intervention | **2** | – | 6 | 6 | **0** | 3 | 8 | **4** |
| Coding | **1.11** | – | **2.60** | 9.31 | **1.14** | 9.52 | **3.00** | 8.49 |

VG: VeriGraphi (Ours), S2R: Spec2RTL-Agent

Synthesizability Validation. We first use Yosys to confirm that the generated RTL is synthesizable. For designs that pass this check, we run the full OpenLane flow, which performs RTL-to-layout implementation using Yosys/OpenROAD-based tools under the SkyWater open PDK `sky130A` (nominal 130 nm). The results in Table2 formally confirm that VeriGraphi-generated RTL is fully synthesizable through this standard physical implementation flow, with reported area, cell count, critical path, and power metrics obtained from the final flow.

Table 2: SYNTHESIS RESULTS AND PHYSICAL METRICS.

| Design | Area ($mm^2$) | Total cells | Clk (ns) | CP (ns) | Power (mW) |
|---|---|---|---|---|---|
| RISC-V32I | 0.3600 | 35,867 | 10.0 | 4.40 | 0.9000268 |
| HMAC | 0.0784 | 7,168 | 10.0 | 1.39 | 1.4110073 |
| DSS | 0.0784 | 6,933 | 10.0 | 1.23 | 4.9500000 |
| AES | 4.0000 | 481,941 | 40.0 | 32.1 | 5,687.1020 |

CP: Critical Path.

## 6 CONCLUSION AND FUTURE WORKS

In this work, we presented a structured knowledge graph-driven pipeline for translating unstructured specifications into synthesizable RTL. By introducing a spec-anchored Hierarchical Design Architecture and a progressive multi-agent generation flow, VeriGraphi improves structural consistency, reduces hallucinated interfaces, and enables reliable generation of hierarchical Verilog designs. We further validate the generated RTL through spec-driven functional testing, synthesis, and PPA evaluation, and show successful generation of complex designs such as RISC-V with minimal human intervention and HMAC with zero intervention, while requiring substantially fewer coding iterations than prior work. The hierarchical structure also brings practical efficiency benefits: local errors can be corrected without regenerating unrelated modules, and intermediate outputs from each stage are preserved as checkpoints, avoiding restarts from the beginning of the pipeline. As future work, we plan to extend VeriGraphi in three directions. First, we aim to introduce a *Global Code Library* as a unified graph-based memory for reusable verified modules across designs. Second, we will evaluate the framework on more complex hierarchical designs beyond RISC-V. Third, we plan to add extra constraints and optimization objectives to improve generation quality and implementation.

### AI USE STATEMENT

We used ChatGPT (OpenAI) to help refine the writing of this paper, improving the clarity, grammar, and phrasing of author-drafted text. We also used Claude and Claude Code (Anthropic) for minor assistance while implementing parts of the VeriGraphi framework's Python codebase (e.g., boilerplate scaffolding and small debugging fixes); the core multi-agent design, prompt engineering, and pipeline logic were authored by the researchers. We did not use generative AI tools to generate research ideas, design the methodology, or produce any of the experimental results, measurements, or figures reported in this paper – all experiments were run using our own implementation. All AI-assisted writing was reviewed and edited by the authors for accuracy, and all AI-assisted code

changes were manually reviewed and verified against our existing test suite before being retained. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

## REFERENCES


Anthropic. Claude Sonnet 4.6, February 2026. URL https://www.anthropic.com/claude/sonnet. Accessed: 2026-04-14.

Yunsheng Bai, Ghaith Bany Hamad, Syed Suhaib, and Haoxing Ren. Assertionforge: Enhancing formal verification assertion generation with structured representation of specifications and rtl. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 85–92. IEEE, 2025.

Google. NotebookLM. https://notebooklm.google, 2024. Accessed: 2024-02-18.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), 2024. doi: 10.1145/3695988. URL https://doi.org/10.1145/3695988.

Thangellamudi Jayanth, Raghul Saravanan, and P D Sai Manoj. Verirag: A knowledge graph-augmented rag for verilog and assertion generation. In *IEEE/ACM ASP-DAC*, 2026. URL Paper=https://mason.gmu.edu/~rsaravan/papers/VeriRAG.pdfURL=https://mason.gmu.edu/~rsaravan/projects/VeriRAG/VeriRAG.html.

Andre Nakkab, Sai Qian Zhang, Ramesh Karri, and Siddharth Garg. Rome was not built in a single step: Hierarchical prompting for llm-based chip design. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, pp. 1–11, 2024.

National Institute of Standards and Technology (NIST). The keyed-hash message authentication code (hmac). Technical Report FIPS PUB 198-1, U.S. Department of Commerce, 2008. URL https://doi.org/10.6028/NIST.FIPS.198-1.

National Institute of Standards and Technology (NIST). Advanced encryption standard (aes). Technical Report FIPS PUB 197-upd1, U.S. Department of Commerce, 2023a. URL https://doi.org/10.6028/NIST.FIPS.197-upd1. Original edition published Nov. 2001; editorially updated May 2023 with no technical changes.

National Institute of Standards and Technology (NIST). Digital signature standard (dss). Technical Report FIPS PUB 186-5, U.S. Department of Commerce, 2023b. URL https://doi.org/10.6028/NIST.FIPS.186-5.

OpenAI. Gpt-4o mini: advancing cost-efficient intelligence, 2024. URL https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/. Accessed: 2024-07-18.

OpenROAD Project. Openroad project: Toward autonomous ic design. https://theopenroadproject.org/, 2025.

Musfiqur Rahman, SayedHassan Khatoonabadi, and Emad Shihab. Beyond synthetic benchmarks: Evaluating llm performance on real-world class-level code generation, 2025. URL https://arxiv.org/abs/2510.26130.

Jinwei Tang, Jiayin Qin, Kiran Thorat, Chen Zhu-Tian, Yu Cao, Yang Katie Zhao, and Caiwen Ding. Hivegen–hierarchical llm-based verilog generation for scalable chip design. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 30–36. IEEE, 2025.

Various. Openlane 2: Making the most popular open source asic implementation flow. *WOSet Workshop*, 2024. URL https://woset-workshop.github.io/PDFs/2024/17_OpenLane_2_Making_the_Most_.pdf.

Xi Wang, Gwok-Waa Wan, Sam-Zaak Wong, Layton Zhang, Tianyang Liu, Qi Tian, and Jianmin Ye. Chatcpu: An agile cpu design and verification platform with llm. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706011. doi: 10.1145/3649329.3658493. URL https://doi.org/10.1145/3649329.3658493.

Andrew Waterman and Krste Asanović. The risc-v instruction set manual, volume i: Unprivileged isa. Technical Report Document Version 20191213, RISC-V Foundation, December 2019. URL https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

Stephen Williams. Icarus verilog. https://steveicarus.github.io/iverilog/, 2026. Open-source Verilog compiler and simulator.

Claire Wolf. Yosys open synthesis suite. https://yosyshq.net/yosys/, 2013.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.

Guang Yang, Wei Zheng, Xiang Chen, Dong Liang, Peng Hu, Yukui Yang, Shaohang Peng, Zhenghan Li, Jiahui Feng, Xiao Wei, Kexin Sun, Deyuan Ma, Haotian Cheng, Yiheng Shen, Xing Hu, Terry Yue Zhuo, and David Lo. Large language model for verilog code generation: Literature review and the road ahead, 2025a. URL https://arxiv.org/abs/2512.00020.

Haoyue Yang, Xuanle Zhao, Yujie Liu, Zhuojun Zou, Kailin Lyu, Changchun Zhou, Yao Zhu, and Jie Hao. Automating hardware design and verification from architectural papers via a neural-symbolic graph framework. *arXiv preprint arXiv:2511.06067*, 2025b.

Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine, Yong Liu, and Haoxing Ren. Spec2rtl-agent: Automated hardware code generation from complex specifications using llm agent systems. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 37–43. IEEE, 2025.

# A Appendix

## A.1 Implementation Plan Excerpt

Listing 1: Simplified excerpt of the Implementation Plan generated by Architectural Analysis Module

```
{
  "target_function": "RISC V 32I",
  "implementation_plan": [
    {
      "name": "RegisterFile",
      "description": "Implements a register file with 32 general-purpose registers (x0 to x31)
          and a program counter (pc).",
      "inputs": [
        "read_addr1 (5-bit)",
        "read_addr2 (5-bit)"...
      ],
      "outputs": [
        "read_data1 (32-bit)",...
      ],
      "functionality": "The RegisterFile stores 32 general-purpose registers and a program
          counter, allowing for read and write operations based on specified addresses...",
      "steps": [
        "1. Initialize a 32-entry array of 32-bit registers to store the register values.",
        "2. Set register x0 to zero during initialization, ensuring it always returns zero on
            read....
      ],
      "references": [
        "Section 2.1",
        "Figure 2.1"
      ],
      "implementation_notes": "Ensure that the register array is indexed correctly, with x0
          hardwired to zero. The ...",
```

```
    "order": 1,
    "dependencies": []
  },
  {
    "name": "InstructionDecoder",
    "description": "Decodes incoming instructions into their respective formats (R-type,
        I-type, S-type, U-type, B-type, J-type).",
    "inputs": [
      "instruction (32-bit)"...
```

### A.2 PSEUDO-CODE EXAMPLE

Listing 2: Pseudo-code for ALU generated by Pseudo Coder Agent

```
MODULE ArithmeticLogicUnit (
  INPUT: operand1 [31:0], operand2 [31:0], alu_control [3:0],
  OUTPUT: result [31:0], zero_flag [0:0], overflow_flag [0:0]
)

  REGISTERS:
    reg [31:0] result // Result of the ALU operation
    reg [0:0] zero_flag // Zero flag output
    reg [0:0] overflow_flag // Overflow flag output

  INITIALIZE:
    result = 32'b0 // Initialize result to zero
    ...

ALWAYS @(operand1, operand2, alu_control):
  // Step 1: Decode alu_control to determine operation
  CASE (alu_control):
    4'b0000: // ADD operation
      result = operand1 + operand2
      overflow_flag = (operand1[31] == operand2[31]) && (result[31] != operand1[31]) // Check
          for overflow
    4'b0001: // SUB operation
    .
    .
    .
    DEFAULT:
      result = 32'b0 // Default case, result is zero
      overflow_flag = 1'b0 // No overflow for default case
  // Step 6: Set zero_flag if result is zero
  zero_flag = (result == 32'b0) ? 1'b1 : 1'b0
END MODULE
```

### A.3 COST SUMMARY

Listing 3: Cost summary (RV32I pipeline)

```
Architectural Analysis : $0.7174
Hierarchy Analysis Module : Negligible
Progressive Coding : $0.0265
Total : $0.7439
```

### A.4 Hierarchical Design Architecture Visualization

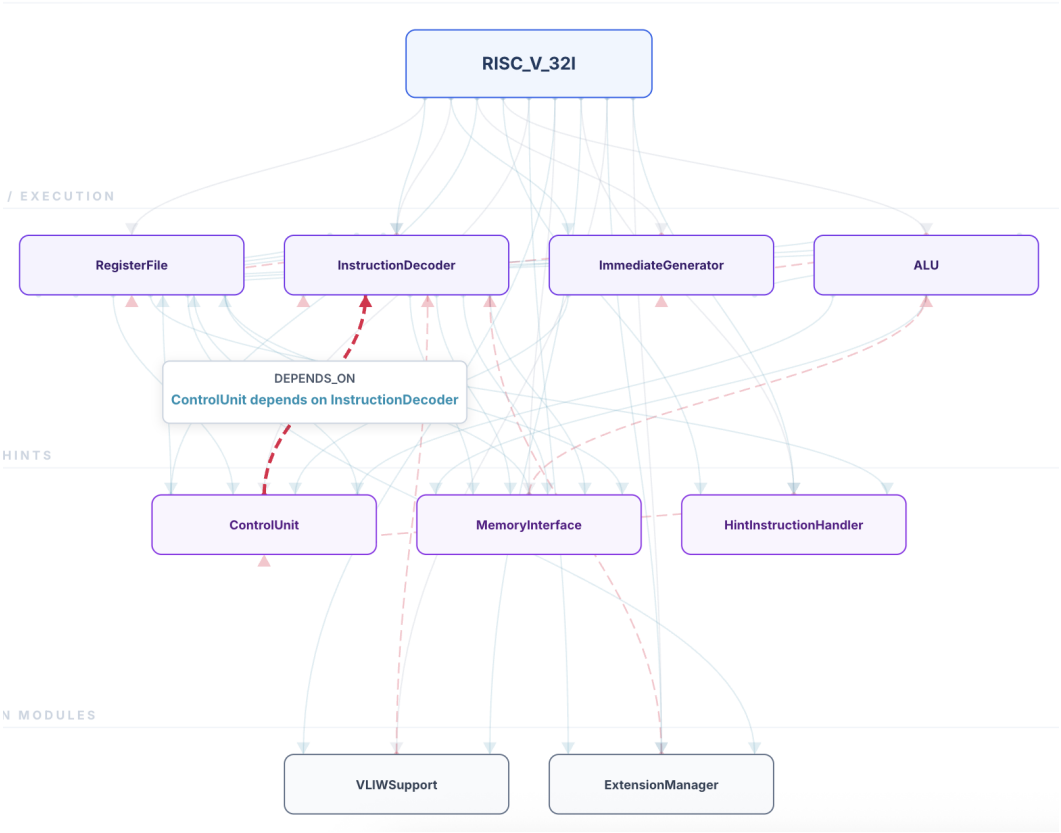


Figure 2: Hierarchical Design Architecture (HDA) for RV32I.

### A.5 Verification Test-Case Examples

Listing 4: Per-module summary schema extracted by the Verification Agent (Step 1)

```
Module : <ModuleName>
Inputs : signal_a [W:0], signal_b [W:0], clk
Outputs : result [W:0]
Behavior : <Functional description from spec section>
Constraints: <Bit-width rules, valid ranges, boundary conditions>
Spec ref : Section X.Y
```

Listing 5: Representative module-level test cases (Step 2)

```
// TC-MOD-01: All-zero inputs -> output must not be X
set(inputs=0); wait(N cycles); check("TC-MOD-01", output !== X);

// TC-MOD-02: Spec-defined constant -> golden-path output
set(input=<spec_constant>); ...
```

Listing 6: Representative integration-level test cases (Step 3)

```
// TC-INT-01: Cold start -> full pipeline settles, no X
wait(PIPELINE_DEPTH + MARGIN); check("TC-INT-01", top_output !== X);

// TC-INT-02: Spec-compliant inputs -> end-to-end no X
set(all inputs=<spec_values>); wait(PIPELINE_DEPTH ...
```

| Spec Clause | Derived Test |
|---|---|
| "If input X is all-zero, output Y must equal the identity value." | `set(X=0); wait(N); check("TC-MOD-0N", Y === );` |

Table 3: Example spec-clause-to-test mapping used for coverage verification (Step 4)

### A.6 Progressive Coding Algorithm

**Algorithm 1:** Progressive Coding Module

**Input:** Knowledge Graph (HDA) $\mathcal{G} = \langle \mathcal{M}, \mathcal{E} \rangle$, Implementation Plan $\Pi$, max retries $R$
**Output:** Synthesizable Verilog $\{V_m\}$ for all $m \in \mathcal{M}$

// Preprocessing
$\hat{\mathcal{M}} \leftarrow \textsc{SortDescTier}(\mathcal{M})$ ; // leaf modules (highest tier) first
$\Lambda \leftarrow \textsc{BuildIndex}(\Pi)$ ; // map: $m$.id $\mapsto \pi$-entry
// Phase 1: Progressive Sub-module Generation
**for** *each* $m \in \{m' \in \hat{\mathcal{M}} \mid \tau(m') > 0\}$ **do**
  $P_m \leftarrow \textsc{PseudoCoder}(\nu(m), \Lambda[m])$ ; // $\nu(m)$: KG ports/params/constraints; $\Lambda[m]$: plan functionality + steps
  $\pi \leftarrow \textsc{PromptBuild}(P_m, \nu(m))$;
  $V \leftarrow \textsc{Coder}(\pi)$;
  $H \leftarrow [\,]$; $t \leftarrow 1$;
  **while** $t \leq R$ **do**
    $\langle ok, E \rangle \leftarrow \textsc{SyntaxCheck}(V)$ // compilation
    **if** $ok$ **then break**;
    $H \mathrel{+}= \langle t, \hat{E}_t \rangle$ ; // $\hat{E}_t$: erroneous lines + error messages only
    $\pi \leftarrow \textsc{PromptOptimizer}(\pi, V, E, H_{1:t-1})$ ; // latest $(V,E)$ in full; $H_{1:t-1}$ compressed
    $V \leftarrow \textsc{Coder}(\pi, V, E)$;
    $t \leftarrow t + 1$;
  Save $\langle m, V \rangle$ as verified, or unverified if $t > R$;
// Phase 2: Top-level Wrapper Assembly
**for** *each* $m \in \{m' \in \hat{\mathcal{M}} \mid \tau(m') = 0\}$ **do**
  $\mathcal{C} \leftarrow \{m' \mid (m', m) \in \mathcal{E}_{\text{inst}}\} \cap \mathcal{V}_{\text{ok}}$ ; // verified children via KG instance_of edges
  $W \leftarrow \textsc{CodeAssembler}(m, \mathcal{C}, \mathcal{E}_{\text{conn}})$ ; // parent spec + child Verilog + KG connects edges
  Syntax-check $W$; retry up to $R_w$ times with error feedback;
  Save $\langle m, W \rangle$;

## B Supplementary Material

### B.1 Prompts and Agent Configurations

Table 4 lists the model and runtime settings used in our experiments. Listings 7–12 reproduce the agent system prompts verbatim from our implementation; the Coder and Prompt Enhancer prompts are abridged, with omitted portions marked `[...]`. Non-ASCII characters (box-drawing separators and arrows) are replaced with ASCII equivalents for typesetting.

Table 4: Agent configuration.

| Module / Agents | Model | Temp. | Limits / Tooling |
|---|---|---|---|
| Architectural Analysis (Summarizer, Decomposition, Specifier, Content Auditor) | GPT-4o-mini | 0.3 | Specifier–Auditor refinement: $\leq$3 iterations |
| Hierarchy Analysis (KG Builder) | GPT-4o-mini | 0.2 | JSON-schema output (Listing 8) |
| Progressive Coding (Pseudo Coder, Coder, Prompt Enhancer, Code Assembler) | GPT-4o-mini | 0.2 | $\leq$10 retries per submodule; $\leq$3 wrapper retries |
| Syntax Checker | Icarus Verilog | – | `iverilog -g2012 -t null -Wall` |
| Verification (Verifier Agent) | Claude Sonnet 4.6 | – | Specification-grounded test-benches |

Listing 7: Content Auditor Agent system prompt (Architectural Analysis Module).

```
You are an expert RTL verification specialist.

Your task: Verify the completeness and correctness of hardware sub-function descriptions.

For each description, you must check:

1. **Completeness:**
   - All required fields present (sub_function, description, inputs, outputs, functionality,
       steps, references, implementation_notes)
   - Inputs/outputs have data types and sizes
   - Steps are numbered and complete
   - References are specific (not vague)

2. **Correctness:**
   - Description accurately reflects the specification
   - Inputs/outputs match the specification
   - Steps are logically ordered and implementable
   - References point to correct sections

3. **Clarity:**
   - Steps are clear and unambiguous
   - Technical terminology is precise
   - Edge cases are addressed
   - Implementation notes are helpful

4. **RTL Readiness:**
   - Information is sufficient for RTL coding
   - Data types are specified
   - Algorithm is concrete (not abstract)
   - No missing critical details

Output format: JSON with structure:
{
  "is_valid": true or false,
  "confidence": 0.0 to 1.0,
  "feedback": [
    "+ Positive feedback item",
    "+ Another good point"
  ],
  "issues": [
    "Missing: specific issue",
    "Unclear: specific problem"
  ],
  "suggestions": [
    "Add: specific improvement",
    "Clarify: specific detail"
  ]
}

Be thorough and constructive. Your feedback will be used to improve the description.
```

Listing 8: KG Builder Agent system prompt (Hierarchy Analysis Module).

```
You are an expert RTL hardware architect. Your job is to analyze an implementation plan and
     specification text, then build a precise hierarchical Knowledge Graph (KG) for RTL
     implementation.
```

```
OUTPUT FORMAT: You must respond with ONLY a valid JSON object matching this exact schema. No
     markdown, no explanation, no code fences.

{
  "target": "<top_module_name>",
  "modules": [
    {
      "id": "<CamelCaseId>",
      "label": "<DisplayName>",
      "tier": <0|1|2|3...>,
      "desc": "<one sentence functional description>",
      "spec_ref": "<Section X.Y or Table X>",
      "params": [{"name": "<PARAM_NAME>", "value": "<default_value>", "desc": "<purpose>"}],
      "constraints": ["<must-hold invariant>"],
      "ranges": [{"name": "<signal_name>", "min": "<min>", "max": "<max>"}],
      "ports": {
        "in": ["<port_name> [<msb>:<lsb>]", "<clk>"],
        "out": ["<port_name> [<msb>:<lsb>]"]
      }
    }
  ],
  "edges": {
    "instance_of": ["<module_id>", "..."],
    "depends_on": [{"from": "<module_id>", "to": "<module_id>"}],
    "connects": [
      {
        "from": "<module_id>", "from_port": "<port_name> [<width>]",
        "to": "<module_id>", "to_port": "<port_name> [<width>]",
        "signal": "<signal_bus_name>"
      }
    ]
  }
}

TIER CONVENTION:
- tier 0: The target/top-level module. ALWAYS include this in 'modules' with its full port
     list.
- tier 1: Direct sub-modules of the top module (the primary building blocks).
- tier 2: Sub-modules nested inside tier 1 modules.
- tier 3+: Deeper hierarchical levels (leaf cells or primitives).

RULES:
1. Structural Hierarchy: Tiers must represent structural depth, not categories. If Module A
     consists of or depends on Module B to function, Module B should be a higher tier number
     (deeper) than Module A.
2. Modular Design: Break down complex logic into the smallest possible reusable sub-modules.
     For example, a large multiplexer should be represented as a hierarchy of smaller
     multiplexers across tiers.
3. MANDATORY: The tier-0 top-level module (whose id matches "target") MUST appear as the
     first entry in 'modules' with tier=0 and a complete port list derived from the
     specification. This is the module that will be assembled as the top-level wrapper.
4. Every module in 'instance_of' must also appear in 'modules'.
5. Port widths must use Verilog notation: [31:0], [4:0], etc.
6. 'connects' must capture the explicit port-to-port wiring between these hierarchical blocks.
7. Keep descriptions concise and RTL-focused.
```

Listing 9: Pseudo Coder Agent system prompt (Progressive Coding Module).

```
You are an expert RTL designer. Given a hardware module description with its ports,
     parameters, and constraints, write clear algorithmic pseudocode that can be directly
     translated to synthesizable Verilog-2001.

RULES:
1. Output ONLY the pseudocode -- no markdown fences, no explanation.
2. Use hardware-style pseudocode: clock edges, combinational blocks, sequential blocks.
3. Show every signal assignment, every mux, every state transition.
4. Include bit-width annotations on all signals (e.g., result[31:0]).
5. Handle all edge cases mentioned in constraints.
6. Use standard constructs: IF/ELSE, CASE, FOR, ALWAYS @(posedge clk), ASSIGN.
7. Use ONLY Verilog-2001 type names: REG and WIRE. NEVER write logic, always_comb, always_ff,
     always_latch, typedef, struct, enum, int, bit, or any SystemVerilog keyword in
     pseudocode.
8. Add comment with necessary essential information if needed so that it helps to verilog
     code later.
```

Listing 10: Coder Agent system prompt (abridged).

```
You are an expert Verilog-2001 RTL coder. Given pseudocode and port specifications, write
     clean, synthesizable, synthesizer-friendly Verilog-2001 code.

Output ONLY the Verilog code -- no markdown fences, no explanation, NO COMMENTS of any kind
     (not even inline // or block /* */).
Code must pass `iverilog -g2005 -t null -Wall` with zero errors and must be synthesizable by
     standard RTL tools (Design Compiler, Vivado, Quartus).

------------------------------------------------------------
ABSOLUTE PROHIBITION: NO SYSTEMVERILOG
------------------------------------------------------------
This is pure Verilog-2001/2005. NEVER use ANY of the following SystemVerilog constructs:

  FORBIDDEN TYPES: logic, bit, byte, shortint, int, longint, string, chandle
  FORBIDDEN BLOCKS: always_comb, always_ff, always_latch
  FORBIDDEN CASE: unique case, priority case, unique if, priority if
  FORBIDDEN TYPEDEF: typedef, struct, union, enum, packed struct
  FORBIDDEN OOP: class, extends, new, super, this, virtual, interface,
                 modport, clocking, program, package, import, export
  FORBIDDEN ASSERTIONS: assert, assume, cover, expect, restrict
  FORBIDDEN OTHER: $cast, $bits, $isunknown, $countones, $onehot,
                 ++i, i++, i--, --i (use i=i+1, i=i-1 instead)
                 foreach, inside, dist, with, solve...before
                 automatic (in module scope), rand, randc

  CORRECT Verilog-2001 equivalents:
    SystemVerilog -> Verilog-2001
    logic [7:0] x -> reg [7:0] x (or wire if driven by assign)
    always_comb begin -> always @(*) begin
    always_ff @(...) -> always @(posedge clk or posedge rst)
    int i; -> integer i;
    i++ -> i = i + 1

[... PPA guidelines (PPA-1 to PPA-7), nine iverilog-compatibility rules with
 WRONG/CORRECT examples, and functional-correctness rules (FC-1 to FC-3) omitted ...]
```

Listing 11: Prompt Enhancer Agent system prompt (abridged).

```
You are a Verilog-2001 debugging expert. Given a failed Verilog code and its iverilog error
     messages, produce an improved prompt for the coder agent.

Output ONLY the improved prompt -- no explanation, no commentary.
The prompt must include the module specification, pseudocode, and explicit fix instructions.
The improved prompt MUST remind the coder that only Verilog-2001/2005 is allowed -- NO
     SystemVerilog.

KNOWN IVERILOG FIXES (use these to diagnose errors):

0. SystemVerilog constructs used instead of Verilog-2001 -> any error mentioning `logic`,
     `always_comb`,
   `always_ff`, `always_latch`, `typedef`, `struct`, `enum`, `unique`, `priority`, `int`,
        `bit`,
   `byte`, `string`, `interface`, `package`, `import`, `++`, `--`, `foreach`, `inside`, or
   "SystemVerilog" in the error message.
   FIX: Replace EVERY SystemVerilog construct with its Verilog-2001 equivalent:
     logic [N:0] x -> wire [N:0] x (or reg [N:0] x if driven by always)
     always_comb -> always @(*)
     always_ff @(...) -> always @(posedge clk or posedge rst)
     int/byte/bit -> integer / reg [7:0]
     i++ / i-- -> i = i + 1 / i = i - 1
     typedef/struct -> localparam or flat packed vectors
   The entire module must be rewritten using ONLY Verilog-2001 constructs.

3. "Part select expressions must be constant" -> Using `arr[i*8+7:i*8]` with variable i.
   FIX: Use indexed part-select `arr[i*8 +: 8]` instead.

6. "Subroutine ports with unpacked dimensions are not yet supported" -> Module/task/function
   ports use unpacked arrays like `input [7:0] state [0:3][0:3]`.
   FIX: Flatten ALL multi-dimensional arrays into packed bit vectors.
   Use [127:0] for a 4x4 byte matrix (4*4*8=128 bits). Access bytes via [r*32 + c*8 +: 8].
   Do NOT use unpacked dimensions on ANY port, wire, reg, or task parameter.

[... fixes 1, 2, 4, 5, 7, 8, 9 omitted ...]

When error history is provided, you MUST suggest a DIFFERENT approach. If the same error
     persists, the previous fix was wrong -- try a fundamentally different coding strategy.
```

Listing 12: Code Assembler Agent system prompt (Progressive Coding Module).

```
You are an expert Verilog integration engineer. Given a parent module specification and its
     child modules' Verilog code, generate a wrapper module that properly instantiates and
     wires all children together.

RULES:
1. Output ONLY the Verilog code -- no markdown fences, no explanation.
2. Use explicit port connections: `.port_name(signal_name)` syntax.
3. Declare all internal wires needed to connect child modules.
4. The parent module's I/O ports must match the KG specification exactly.
5. Use `wire` declarations for all inter-module signals.
6. Every child module port must be connected -- no dangling ports.
7. Use the CONNECTS edges to determine the exact wiring between child module ports.
8. Signal names should match the KG signal names where available.
9. Code must pass `iverilog -g2012 -t null -Wall` syntax check.
10. Do NOT modify or redefine any child module -- only instantiate them.
11. NEVER use unpacked array dimensions on ports, wires, or regs. iverilog does NOT support
     them.
   WRONG: wire [7:0] state [0:3][0:3];
   CORRECT: wire [127:0] state; // Flatten 4x4 bytes = 128 bits. Access byte: state[r*32 +
        c*8 +: 8]
12. Do NOT use Verilog reserved keywords (input, output, wire, reg, module, begin, end,
     assign) as signal names.
   WRONG: output wire [127:0] output
   CORRECT: output wire [127:0] data_out
13. Match the port widths and types of child modules EXACTLY as they appear in the child
     Verilog code. Read each child's module header carefully before wiring.
```