LLM-FSM: Scaling Large Language Models for Finite-State Reasoning in RTL Code Generation

Yuheng Wu
Stanford University
Stanford, CA, USA
yuhengwu@stanford.edu

Berk Gokmen
Stanford University
Stanford, CA, USA
bgokmen@stanford.edu

Zhouhua Xie
Stanford University
Stanford, CA, USA
xzh015@stanford.edu

Peijing Li
Stanford University
Stanford, CA, USA
peli@stanford.edu

Caroline Trippel
Stanford University
Stanford, CA, USA
trippel@stanford.edu

Priyanka Raina
Stanford University
Stanford, CA, USA
praina@stanford.edu

Thierry Tambe
Stanford University
Stanford, CA, USA
ttambe@stanford.edu

Abstract

Finite-state reasoning, the ability to understand and implement state-dependent behavior, is central to hardware design. In this paper, we present LLM-FSM, a benchmark that evaluates how well large language models (LLMs) can recover finite-state machine (FSM) behavior from natural-language specifications and translate it into correct register transfer-level (RTL) implementations. Unlike prior specification-to-RTL benchmarks that rely on manually constructed examples, LLM-FSM is built through a fully automated pipeline. LLM-FSM first constructs FSM with configurable state counts and constrained transition structures. It then prompts LLMs to express each FSM in a structured YAML format with an application context, and to further convert that YAML into a natural-language (NL) specification. From the same YAML, our pipeline synthesizes the reference RTL and testbench in a correct-by-construction manner. All 1,000 problems are verified using LLM-based and SAT-solver-based checks, with human review on a subset. Our experiments show that even the strongest LLMs exhibit sharply declining accuracy as FSM complexity increases. We further demonstrate that increasing test-time compute improves performance on this task, and that LLM-FSM remains extensible by allowing its FSM complexity to scale with future model capabilities.

1 Introduction

Large language models (LLMs) have demonstrated strong reasoning capabilities in tasks such as math problem solving and code generation [13]. Recently, there has been growing interest in applying LLMs to assist electronic design automation (EDA) tasks [9, 36]. A representative example is register transfer-level (RTL) code generation, where a model receives a natural-language (NL) design specification and generates the corresponding implementation [16, 19, 20, 26]. NL is the default interface in LLM-based design workflows, since users describe intended behavior in NL rather than formal specification formats. To improve the performance of LLMs on such NL specification-to-RTL tasks, researchers have explored techniques such as supervised fine-tuning (SFT) [17], reinforcement learning (RL) [42], and multi-agent collaboration [51]. All these approaches can be viewed through the lens of *scaling*, which broadly refers to allocating more computation, during training or inference, to enhance a model's RTL reasoning performance.

In evaluating LLMs' ability to generate RTL, an important component is their finite-state reasoning capability, which refers to the ability to understand and implement state-dependent behavior. This capability underlies a wide range of hardware workflows, including controllers, protocols, and multi-cycle sequencing logic. However, there is no benchmark designed specifically to evaluate LLMs on finite-state reasoning. Existing specification-to-RTL datasets [16, 19, 20, 26–28] contain only a small number of such tasks, and each example requires domain experts to manually write the specification, reference RTL, and testbench, making them difficult to scale. Therefore, our research question is: *How can we automatically create a large and scalable NL specification-to-RTL benchmark that evaluates LLMs on finite-state reasoning?*

In this paper, we present LLM-FSM, a large-scale NL specification-to-RTL benchmark designed to evaluate finite-state reasoning in LLMs. Unlike existing benchmarks that rely on manual construction, our approach fully automates the dataset curation process. The resulting benchmark is both scalable and controllable, enabling new problems to be generated at configurable levels of finite-state machine (FSM) complexity. All reference RTL implementations and testbenches are produced in a correct-by-construction manner. The dataset contains problems spanning diverse and realistic finite-state reasoning scenarios, providing a challenging and extensible benchmark for evaluating LLM-based RTL generation.

As illustrated in Figure 1, the data curation process begins by sampling an **abstract FSM graph** based solely on the specified number of states and a set of structural constraints. These constraints ensure that all states are reachable from reset, transitions form a connected directed graph, and the out-degree and back-edge probabilities lie within prescribed bounds. This graph encodes only the topology of the machine and carries no application-level meaning. An LLM then interprets this abstract graph and generates an application context consistent with its transition structure, assigning **semantic** roles to each state and defining the inputs, outputs, and conditions associated with each transition. This FSM description is stored in a **structured FSM YAML format**. We check that the FSM YAML representation is isomorphic to the original abstract graph and discard any samples that fail this check. Using the fsm2sv [3] tool, we translate the full FSM YAML into a SystemVerilog implementation. We additionally design a generator that constructs a testbench by systematically traversing the transitions encoded in the FSM YAML. Together, these steps yield the **reference RTL implementation** and its corresponding **testbench**.

In the next stage, we use an LLM to translate the FSM YAML into an **NL specification** that describes the behavior of the machine. Given the state mappings contained in the YAML, the LLM is then asked to reconstruct the FSM by converting the specification back into YAML format. Using the fsm2sv tool once again, we generate SystemVerilog code from both the **reconstructed FSM** and the original FSM. The two designs are then passed to Yosys [47] for SAT-solver-based equivalence checking to determine whether the specification preserves the behavior of the original FSM. Specifications that fail the equivalence check are discarded, and a subset of the passing specifications undergoes human review to verify narrative clarity and hardware plausibility of the NL description.

Our curated LLM-FSM dataset contains 1,000 problems, grouped into three difficulty tiers (easy, medium, and hard) based on state number and edge complexity. As shown in Figure 2, we evaluate models under three pipelines: (1) **Specification→RTL**, which tests end-to-end Verilog generation; (2) **Specification→YAML→RTL**, where the model constructs the FSM in a structured YAML format and RTL is generated using fsm2sv; and (3) **Specification→SystemC**, where the model directly produces SystemC for validation through SystemC-Verilog co-simulation (Co-Sim). Across these settings, we evaluate a wide range of frontier models and find that even the most advanced LLMs struggle on the hard tier of LLM-FSM.

Our results show that LLM-FSM exposes clear limitations of current LLMs in finite-state reasoning and offers a challenging benchmark for evaluating future models. We further demonstrate that test-time scaling yields consistent performance improvements. In summary, our work makes the following contributions:

- An automatic pipeline that synthesizes scalable RTL benchmarks, verified through a SAT solver, and designed to evolve jointly with advances in LLM capabilities;
- A rigorous evaluation across three tool-chain pipelines, demonstrating that existing models struggle on LLM-FSM and underscoring the benchmark's challenging nature;
- An analysis of parallel test-time compute, demonstrating that parallel sampling outperforms serial decoding for finite-state reasoning.

2 Related Work

In this section, we review related work in three areas: the application of LLMs to RTL code generation, existing benchmarks for evaluating RTL code generation tasks, and recent advances in scaling LLMs to enhance reasoning capabilities.

2.1 LLMs for RTL Code Generation

Researchers have explored using LLMs for hardware code generation since the rise of LLMs [9, 36]. However, due to the scarcity of hardware description language (HDL) data compared to high-level languages such as Python or C++ [35], pretrained models often perform poorly on hardware-related tasks [18]. To address this limitation, recent studies have focused on SFT of LLMs on domain-specific corpora to better adapt them for RTL design and synthesis [1, 8, 11, 12, 17, 18, 25, 37, 43]. Beyond SFT, some works further employ RL-based post-training [13, 38, 42] to enhance code correctness. At the inference stage, many approaches design workflows incorporate execution feedback [34] and leverage multi-agent collaboration [24, 50, 51] to further enhance the quality and reliability of generated RTL code.

Beyond direct RTL code generation, LLMs have also been explored for higher-level hardware design workflows [31]. Some approaches first generate high-level languages such as C or Python, which are then translated into domain-specific representations [4, 15]. In addition, LLMs are increasingly applied to other stages of the hardware design flow, including testbench generation [5, 21, 29, 30], formal specification synthesis [33, 48], and temporal logic specification generation [10, 22].

Table 1: Comparison of RTL code generation benchmarks. The five dimensions are: (1) *Dataset Size*: the number of test instances included in the benchmark; (2) *Automated Generation*: the dataset is automatically constructed and scalable; (3) *Difficulty Control*: problem difficulty can be adjusted; (4) *Realistic Specification*: tasks are described in NL specifications; and (5) *Automated Verification*: testbenches are automatically generated for validation.

Benchmark	Dataset Size	Automated Generation	Difficulty Control	Realistic Specification	Automated Verification
RTLLM v1 [20]	29	✗	✗	✓	✗
RTLLM v2 [19]	50	✗	✗	✓	✗
VerilogEval v1 [16]	156	✗	✗	✓	✗
VerilogEval v2 [26]	156	✗	✗	✓	✗
ArchXBench [28]	51	✗	✓	✓	✗
CVDP [27]	783	✗	✓	✓	✗
LLM-FSM (Ours)	1000	✓	✓	✓	✓

2.2 RTL Code Generation Benchmarks

As shown in Table 1, several benchmarks have been developed to evaluate LLM performance on RTL code generation. Among them, the two most widely used are RTLLM [19, 20] and VerilogEval [16, 26]. RTLLM contains 50 problems, while VerilogEval includes 156 tasks covering both combinational logic modules and FSMs. ArchXBench [28], on the other hand, consists of 51 human-authored high-level RTL design tasks, such as generating FFT and CNN modules. In all these benchmarks, the specifications and corresponding testbenches are manually written, making them difficult to scale to larger datasets. In addition, RTL-Repo [2] provides a fill-in-the-blank style task focusing on completing partial RTL code, and CVDP [27] offers 783 human-written questions covering the broader RTL design pipeline. Although larger in scale, these datasets are still manually curated and span diverse tasks such as debugging, code comprehension, and design analysis, rather than focusing on specification-to-RTL generation.

2.3 Scaling LLMs for Reasoning

Scaling refers to allocating more computation to enhance the reasoning capabilities of LLMs. Such scaling can occur both during training and at inference time. At the training stage, scaling can be achieved through RL [13] or by SFT on reasoning traces distilled from stronger teacher models [14]. These approaches encourage the model to generate longer and more coherent reasoning chains, resulting in improved performance on complex reasoning tasks.

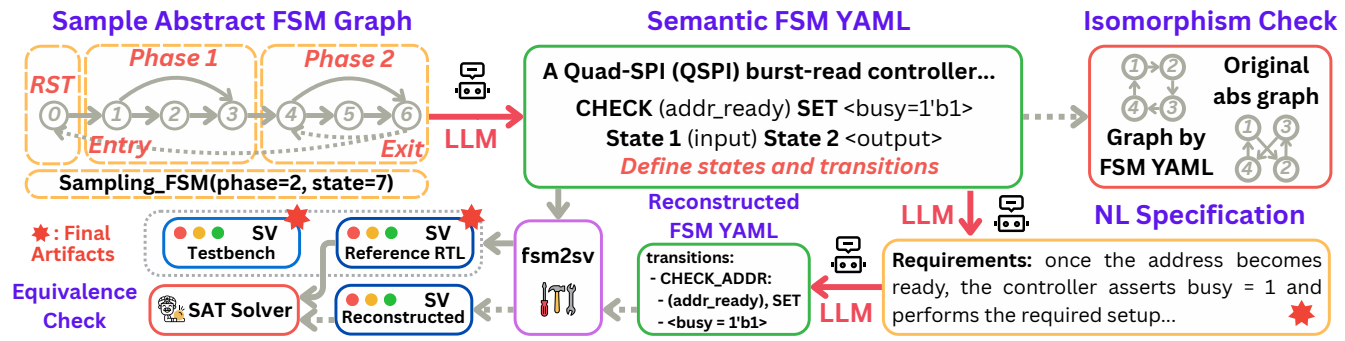

Figure 1: Overview of the LLM-FSM data curation pipeline. The process begins by constructing an abstract FSM graph, followed by LLM-based specification generation, automatic RTL and testbench synthesis, and isomorphism/equivalence check.

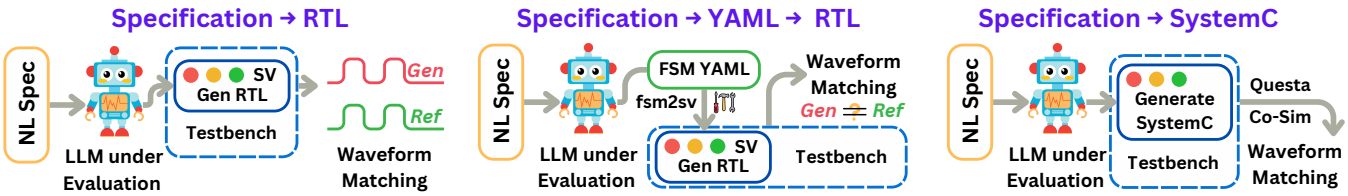

Figure 2: Overview of the LLM-FSM evaluation pipeline. An NL specification is processed through three tool-chain settings: Specification→RTL, Specification→YAML→RTL, and Specification→SystemC. Each model prediction is executed under the same reference testbench, and correctness is determined by cycle-by-cycle output matching against the reference RTL.

At the inference stage, scaling is commonly referred to as test-time scaling (TTS). It can be realized by encouraging deeper and more deliberate reasoning within a single inference path [23, 44]. Alternatively, multi-trace TTS [7, 32, 45, 46] samples multiple candidate completions in parallel and selects the best one using either verifier-based evaluation [39] or voting-based aggregation [40, 41]. Recent studies further integrate search algorithms [6, 49] that interleave generation and selection in a step-by-step manner, refining the output through iterative exploration and verification.

3 Methods

In this section, we describe how LLM-FSM is constructed and how models are evaluated on it. We first synthesize abstract FSM topologies (Section 3.1), enrich them with LLM-generated semantics (Section 3.2), and compile them into RTL and testbenches (Section 3.3), validating consistency through SAT-based equivalence (Section 3.4). We then present dataset statistics, human validation, and the evaluation pipelines used to assess model performance (Sections 3.5–3.7).

3.1 Abstract FSM Graph Construction

Phase-based abstract graph structure. We represent each FSM using a two-level structure organized into phases. A phase corresponds to a coherent stage of operation, such as initialization, data transfer, or error handling. Each phase is a subgraph with a single entry and exit, and all internal states lie on paths between them. Phase transitions are modeled by directed edges from the exit of one phase to the entry of another, allowing the abstract graph to capture high-level control flow.

Topology generation algorithm. For each phase, we generate a minimal chain from entry to exit to ensure reachability, then add forward branches, back edges, and self-loops under user-controlled probabilities while capping the out degree. We add a reset block and connect phases in a simple cycle to guarantee global reachability. This ensures that every sampled abstract FSM is structurally valid. Additional inter-phase edges are sampled to create jumps between phases. This procedure produces an abstract FSM graph whose structure is determined by a small set of topology parameters.

Example. Figure 3 shows an example with two phases. Each phase contains an entry–exit chain with additional forward branches, back edges, and self-loops sampled under the specified probabilities, while graph-level edges form a cycle that guarantees global reachability. This two-level organization is intentional: real hardware controllers typically consist of several semantically coherent phases assembled into a larger control graph. Directly sampling an unconstrained flat FSM makes it difficult for an LLM to assign meaningful roles to states or to construct realistic hardware scenarios. By generating topology at both the phase level and the graph level, we obtain abstract FSMs that are structurally rich yet still amenable to consistent semantic interpretation.

3.2 Semantic FSM Generation and YAML Construction

LLM-based semantic FSM generation. As shown in Figure 3, given an abstract FSM graph from Section 3.1, we use an LLM to turn this purely structural object into a semantic FSM. The prompt exposes the phase structure, the exact edge list, and asks the model to

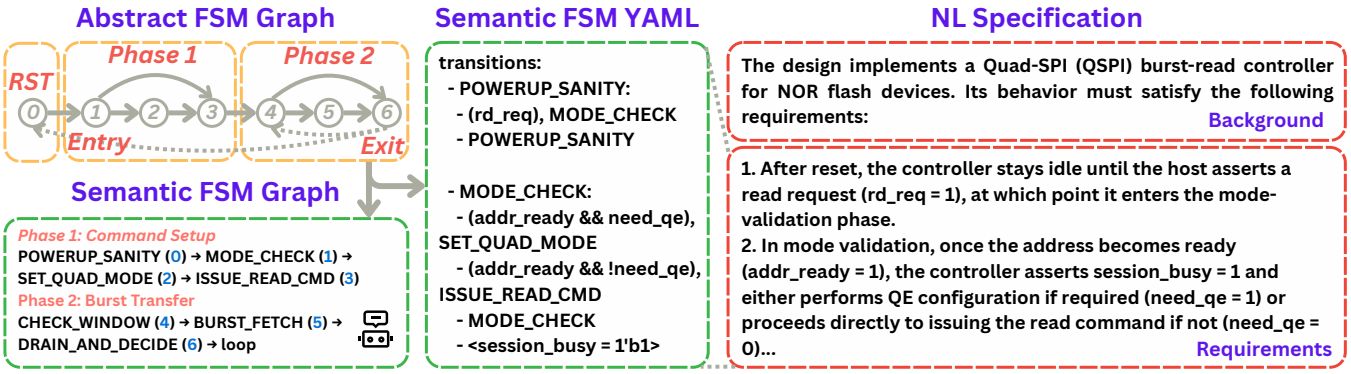

Figure 3: An example illustrating the generation process. The abstract graph is first sampled topologically, and an LLM then assigns semantics, here producing a Quad-SPI burst-read controller for a NOR-flash device.

choose a realistic hardware scenario, assign descriptive names to all states, and design input and output signals. The model then produces an fsm2sv-compatible semantic FSM YAML file that specifies reset behavior, input and output declarations, and for each state, a list of guarded transitions and outputs that follow the provided connectivity. In addition to the YAML, the model also generates a short story of the workflow.

Graph isomorphism verification. As shown in Figure 1, let $G_{\text{abs}} = (V, E)$ denote the abstract FSM graph and let $G_{\text{yaml}} = (\hat{V}, \hat{E})$ be the graph obtained from the transitions in the generated YAML. The state mapping produced by the LLM defines a candidate bijection $f : V \to \hat{V}$, and we require

$$(u, v) \in E \quad \Longleftrightarrow \quad (f(u), f(v)) \in \hat{E}$$

for all $u, v \in V$. Any YAML instance that violates this isomorphism condition is discarded, so the retained semantic FSMs add meaning while preserving the original topology.

3.3 Reference RTL and Testbench Generation

Reference RTL. Given a checked semantic FSM YAML, the reference implementation is generated automatically using the fsm2sv tool, as shown in Figure 1. Because the YAML format fully specifies each state, its outputs, and the ordered conditional transitions, the translation to synthesizable SystemVerilog is mechanical: every YAML transition becomes a guarded branch in the `always_comb` block, and state encodings are assigned in a consistent one-hot or counter style. Since the YAML itself has already passed the topology-preserving isomorphism check, the resulting RTL is correct-by-construction relative to the input FSM.

Testbench generation. The fsm2sv package does not include a testbench generator, so we extend the tool with a testbench synthesis module. The key requirement is to produce a set of input sequences that covers all states and all transitions of the FSM. Let the FSM be a directed graph $G = (V, E)$ with initial state s_0. For every edge $e = (u, v) \in E$, the generator performs:

find a path $s_0 \rightsquigarrow u$, emit inputs that satisfy the guard of e.

This guarantees that each transition is exercised at least once. The generator runs in polynomial time in the size of the FSM. It also emits additional random stimuli to improve robustness. The resulting testbench is a self-contained module that instantiates the DUT, drives the generated input sequences cycle by cycle, and records waveforms for debugging.

3.4 Specification Generation and Formal Verification

NL specification synthesis. As shown in Figure 1, given a topology-verified YAML FSM $\mathcal{Y}$, we first ask an LLM to produce an NL specification $\Sigma = (\Sigma_{\text{IO}}, \Sigma_{\text{req}})$. The prompt exposes only the inputs/outputs, reset configuration, and transition table, and requires: (1) an Inputs and Outputs section (Σ_{IO}) that lists every signal using the exact YAML names; and (2) a Requirements section (Σ_{req}) that paraphrases each group of transitions into requirements. This defines a forward map $F : \mathcal{Y} \to \Sigma$ that hides state names but keeps the transition semantics.

To check that the specification is semantically complete, we perform a second LLM pass that reconstructs a fsm2sv-compatible YAML $\widetilde{\mathcal{Y}}$ from Σ and the state mapping. Invalid or structurally inconsistent reconstructions are discarded, leaving only pairs $(\mathcal{Y}, \widetilde{\mathcal{Y}})$ where both directions $\mathcal{Y} \xrightarrow{F} \Sigma \xrightarrow{G} \widetilde{\mathcal{Y}}$ succeed.

SAT-based equivalence checking. For each structurally correct pair $(\mathcal{Y}, \widetilde{\mathcal{Y}})$, we compile both YAML files with fsm2sv to produce two RTL machines with identical I/O interfaces:

$$\mathcal{M} = (S, s_0, I, O, \delta, \lambda), \qquad \widetilde{\mathcal{M}} = (\widetilde{S}, \widetilde{s}_0, I, O, \widetilde{\delta}, \widetilde{\lambda}).$$

To check whether the NL specification preserves the behavior of the original FSM, we use Yosys's equivalence-checking flow (`equiv_make`, `equiv_simple`, `equiv_struct`, `equiv_status`). This constructs a sequential miter between the two RTL designs and searches for an input sequence under which their outputs diverge.

Formally, let the two machines process the same input sequence $(x_0, \ldots, x_T)$. A mismatch at time t is recorded as

$$d_t = \big(\lambda(s_t, x_t) \neq \widetilde{\lambda}(\widetilde{s}_t, x_t)\big), \qquad D = \bigvee_{t \le T} d_t.$$

The solver asks whether some reachable execution can produce $D = 1$. If a counterexample trace is found, the sample is discarded. If Yosys completes the check without reporting a mismatch, we accept

Table 2: LLM-FSM dataset statistics. Tasks are grouped into three difficulty tiers based on total state count. For each tier, we report dataset size, complexity measures, specification word count, and reference RTL code lines.

Tier	Count	States	Avg. Edges	Avg. Phases	Avg. Spec. Word Count	Avg. Ref. Code Lines
Low	334	4–14	11.95	2.71	922.3	154.8
Medium	333	14–27	32.17	5.24	1380.4	301.8
High	333	27–59	65.39	8.83	1957.6	501.3
Overall	1000	4–59	36.48	5.59	1419.6	319.1

the pair as behaviorally equivalent for all executions explored by the checker. Only examples that pass this equivalence check are kept. This round-trip filter ensures that the NL specification is consistent with the transition semantics encoded in the original YAML FSM.

3.5 Data Curation Dynamics

Our curated LLM-FSM benchmark contains 1,000 problems, partitioned into three difficulty tiers based on the number of FSM states. Summary statistics are provided in Table 2.

Generation runtime. All semantic FSMs and NL specifications are generated using gpt-5 through the OpenAI API. The entire generation stage completes quickly under batch parallelism. The dominant computational cost lies in verification: running Yosys's equivalence check on a single FSM typically takes ~ 30 seconds, making formal checking the primary bottleneck of the pipeline.

Filtering statistics. Our pipeline applies two filters: an isomorphism check between the abstract graph and the LLM-generated YAML, and a equivalence check between the reference and reconstructed RTL. Out of 1,500 generated candidates, 1,411 (94.1%) pass the isomorphism test, and 1,085 (76.9%) also pass RTL equivalence. Equivalence-check pass rates decrease with FSM size (95.7%, 82.1%, 62.4% across the three tiers), but remain sufficiently high to scale further by increasing generation budget or adopting hierarchical generation for larger FSMs. We randomly select 1,000 verified examples to form the final LLM-FSM dataset.

3.6 Human Check

To further ensure data quality, we perform a manual audit on a subset of examples. Each sampled instance is examined along four criteria: (1) *State Coverage*: the specification must describe every YAML state with no missing or spurious behaviors. (2) *Transition Coverage*: every YAML transition must be reflected in the specification, with no extra or altered edges. (3) *Specification-FSM Alignment*: the narrative must allow an unambiguous mapping from each described behavior back to the YAML-specified FSM. (4) *Hardware Plausibility*: state names, signal names, and contextual descriptions must form a coherent and realistic hardware scenario. All 20 inspected samples satisfy these criteria, confirming the semantic consistency between the specification and the underlying FSM.

3.7 Evaluation Pipeline

We evaluate models in three settings. (1) **Spec → RTL:** the model generates SystemVerilog directly from the NL specification, and correctness is determined by cycle-accurate agreement with the reference RTL under the same testbench. (2) **Spec → YAML → RTL:** the model first produces an fsm2sv-compatible YAML FSM, which is compiled to RTL and evaluated using the same criterion. (3) **Spec → SystemC:** the model outputs a SystemC design, which we test via SystemC–SystemVerilog co-simulation in Questa; correctness again requires cycle-by-cycle agreement with the reference RTL.

4 Evaluation

4.1 Experimental Setup

In addition to our LLM-FSM benchmark, we also evaluate on two human-written RTL datasets: VerilogEval v2 [26] and RTLLM v2 [19]. We evaluate a broad set of frontier model families, including gpt-5, Claude-4.5, Gemini-2.5, grok-4, Qwen-3, DeepSeek-V3.1/R1, and Llama-4. These models span both proprietary and open-source families and represent current state-of-the-art LLM systems across a wide range of model sizes. For all models, we set the maximum output token budget to 16,384. Temperature and top-p follow each model's default settings. We report Pass@1 as the primary metric, counting a sample as correct only if the generated RTL compiles and passes the reference testbench.

4.2 Main Results

LLM-FSM is a challenging benchmark. As shown in Figure 3, across 18 frontier models and three evaluation pipelines, the overall average Pass@1 is only 41.1%. Among all evaluated LLMs, Claude-4.5-Sonnet achieves the highest score of 80.3%, yet its performance drops to 65.6% on the hard tier. Because our dataset is generated through a fully automated pipeline, increasing the number of states or transitions can produce harder instances, allowing LLM-FSM to evolve alongside future model improvements.

Different evaluation pipelines lead to sharply different outcomes across models. For example, Gemini-2.5-Pro attains 70.4% on the Spec→YAML→RTL task but only 17.9% on the Spec→SystemC task. Across all evaluated models, the SystemC setting shows the lowest average performance, suggesting that LLMs are less familiar with hardware-compatible high-level languages than with RTL. By contrast, the Spec→RTL and Spec→YAML→RTL pipelines yield similar average accuracies, indicating that modern LLMs are able to perform finite-state reasoning directly in RTL without explicitly reconstructing the FSM structure in YAML.

4.3 Analysis

Scaling trend and difficulty analysis. As shown in Figure 4, within the same family, models benefit from increasing parameter scale. On the difficulty axis, accuracy drops sharply as the number of states and edges increases, with the Spec→SystemC pipeline being the most sensitive. This confirms that our generation pipeline provides fine-grained control over task difficulty.

Correlation with human-written benchmarks. To validate that performance on LLM-FSM reflects real-world RTL generation ability, we compare model accuracy on our dataset with two human-written benchmarks: VerilogEval v2 [26] and RTLLM v2 [19]. As shown in Table 4, model performance exhibits strong positive correlation across datasets. Interestingly, both VerilogEval v2 and RTLLM v2

Table 3: Model accuracy (%) across three RTL generation benchmarks: VerilogEval v2 [26], RTLLM v2 [19], and our LLM-FSM dataset. For each evaluation pipeline, the best-performing model is shown in bold and the second-best is underlined.

Model	Verilog Eval	RT LLM	Spec → RTL				Spec → YAML → RTL				Spec → SystemC				Avg.
			Easy	Med.	Hard	Avg.	Easy	Med.	Hard	Avg.	Easy	Med.	Hard	Avg.	
Llama4-Scout	48.7	36.0	24.0	5.1	0.0	9.7	38.0	4.8	0.3	14.4	0.0	0.0	0.0	0.0	8.0
Qwen3-4B	39.1	30.0	20.7	3.6	0.6	8.3	27.5	3.6	0.6	10.6	39.5	11.7	2.1	17.8	12.2
Qwen3-8B	53.2	38.0	14.1	5.4	2.7	7.4	44.9	12.9	2.1	20.0	61.4	29.7	5.1	32.1	19.8
gpt-5-nano	72.4	54.0	55.4	19.2	4.5	26.4	29.9	5.4	0.6	12.0	58.7	20.4	3.3	27.5	22.0
Llama4-Maverick	59.6	50.0	63.2	27.0	6.3	32.2	63.2	24.0	6.6	31.3	35.0	15.0	5.1	18.4	27.3
Qwen3-14B	59.0	42.0	55.4	26.4	9.0	30.3	62.0	23.4	4.8	30.1	63.5	24.6	10.2	32.8	31.1
gpt-oss-20B	46.8	42.0	61.1	34.8	13.5	36.5	45.8	15.6	3.3	21.6	73.4	37.2	9.3	40.0	32.7
gpt-oss-120B	53.8	44.0	77.5	45.6	19.8	47.7	48.5	26.7	8.7	28.0	74.6	44.7	16.8	45.4	40.4
Qwen3-32B	64.7	52.0	66.5	37.5	14.4	39.5	77.2	41.1	13.5	44.0	68.9	34.2	12.6	38.6	40.7
DeepSeek-R1-0528	72.4	58.0	64.7	41.4	10.8	39.0	68.3	46.5	24.0	46.3	68.0	45.0	4.2	39.1	41.5
Gemini-2.5-Flash	60.3	56.0	76.0	55.9	16.5	49.5	86.5	67.0	44.1	65.9	57.8	8.7	0.0	22.2	45.9
DeepSeek-V3.1-Terminus	69.9	50.0	80.2	54.7	9.0	48.0	75.4	52.0	24.9	50.8	79.9	46.8	1.8	42.9	47.2
Gemini-2.5-Pro	77.6	60.0	77.5	59.8	39.6	59.0	87.4	68.2	55.6	70.4	46.1	7.5	0.0	17.9	49.1
grok-4-fast-reasoning	74.4	58.0	75.1	48.6	37.8	53.9	78.1	50.5	40.8	56.5	74.0	52.6	39.9	55.5	55.5
gpt-5-mini	78.2	54.0	88.6	64.6	41.7	65.0	66.8	36.0	21.6	41.5	84.4	60.1	38.7	61.1	55.9
Claude-4.5-Haiku	75.0	54.0	84.7	53.2	30.9	56.3	86.5	55.3	33.9	58.6	82.3	54.7	27.9	55.0	56.6
gpt-5	**86.2**	**64.0**	93.1	**79.6**	59.8	77.5	93.7	76.6	65.8	78.7	82.0	73.3	**55.0**	70.1	75.4
Claude-4.5-Sonnet	82.1	**64.0**	**95.5**	79.3	**70.0**	**81.6**	**94.0**	**83.2**	**72.1**	**83.1**	**93.4**	**80.8**	54.7	**76.3**	**80.3**
Average	65.2	50.3	65.2	41.2	21.5	42.6	65.2	38.5	23.5	42.3	63.5	35.9	15.9	38.5	41.1

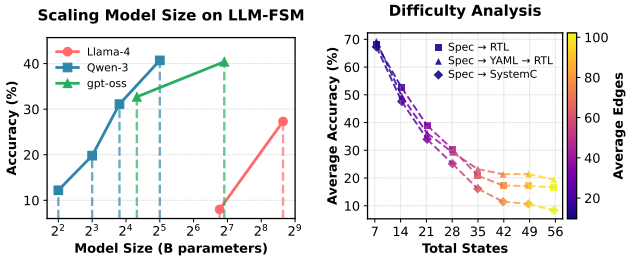

Figure 4: Scaling and difficulty analysis on LLM-FSM. Left: scaling behavior of different model families. Right: accuracy averaged across all models within each difficulty bin.

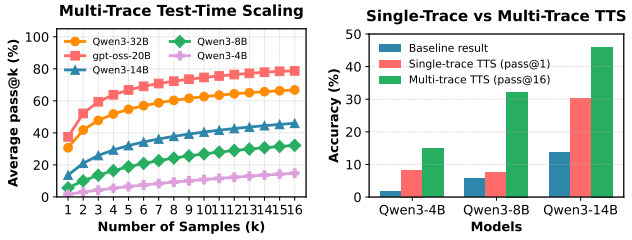

Figure 5: TTS for finite-state reasoning. Left: multi-trace TTS pass@k scaling on LLM-FSM. Right: comparison of single-trace TTS vs multi-trace TTS at $k = 16$.

Table 4: Correlation between performance on our benchmark and human-written datasets. Pipeline 1: Spec→RTL, Pipeline 2: Spec→YAML→RTL, Pipeline 3: Spec→SystemC. We report Pearson (P) and Spearman (S) correlations.

Dataset	Pipeline 1		Pipeline 2		Pipeline 3		Overall Avg.	
	P	S	P	S	P	S	P	S
VerilogEval	0.82	0.85	0.78	0.78	0.66	0.62	0.83	0.87
RTLLM	0.85	0.82	0.85	0.85	0.59	0.52	0.84	0.83

primarily evaluate the direct Spec→RTL generation path, and we observe that correlations are indeed highest for our Pipeline 1 results, further suggesting that LLM-FSM captures the same underlying reasoning skills required for hand-written RTL benchmarks.

Error analysis. We manually analyze a subset of incorrect generations and identify four common failure modes. (1) *Syntax errors*: models occasionally include invalid syntax in generated RTL code that fails to compile. (2) *Incorrect timing semantics*: the implementation violates cycle-level behavior described in the specification, such as transitioning out of a state earlier or later than required. (3) *State or transition mistakes*: the generated code introduces missing, extra, or reordered edges, resulting in an FSM whose structure differs from the intended one. (4) *Formatting errors*: tasks involving YAML or SystemC templates often fail because the model does not precisely follow the required schema, leading to invalid FSM descriptions or modules that cannot be parsed.

4.4 Scaling LLMs for Finite-State Reasoning

TTS for Finite-State Reasoning. As shown in Figure 5 (left), increasing the number of samples per question in our multi-trace TTS setting steadily improves each model's pass@k on LLM-FSM. However, simply letting models think before answering (single-trace TTS) is less effective: Figure 5 (right) shows that Qwen3-14B's pass@1 in thinking mode remains well below its own pass@16 under multi-trace TTS, with smaller models showing the same gap.

5 Conclusion

In this paper, we introduce LLM-FSM, a scalable benchmark for translating NL specifications to RTL. The dataset explicitly targets finite-state reasoning, provides fine-grained difficulty control, and checks consistency between each specification and its RTL implementation through SAT-based verification. Evaluating a wide range of LLMs on 1,000 problems, we find that current models still struggle with temporally precise RTL synthesis. Performance on LLM-FSM also correlates with results on human-written RTL benchmarks, indicating that it captures real-world design challenges. We further show that multi-trace TTS boosts model performance on LLM-FSM. Overall, LLM-FSM provides a scalable foundation for evaluating and advancing finite-state reasoning in LLM-based RTL generation.

References

[1] Mohammad Akyash, Kimia Azar, and Hadi Kamali. 2025. RTL++: Graph-enhanced LLM for RTL Code Generation. *ICLAD* (2025).

[2] Ahmed Allam and Mohamed Shalan. 2024. RTL-Repo: A Benchmark for Evaluating LLMs on Large-Scale RTL Design Projects. *LAD Workshop* (2024).

[3] Mohamed Bamakhrama. 2021. fsm2sv: SystemVerilog FSM Generator. *GitHub* (2021).

[4] Christopher Batten, Nathaniel Pinckney, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2024. PyHDL-Eval: An LLM Evaluation Framework for Hardware Design Using Python-Embedded DSLs. *MLCAD* (2024).

[5] Jitendra Bhandari, Johann Knechtel, Ramesh Narayanaswamy, Siddharth Garg, and Ramesh Karri. 2024. LLM-Aided Testbench Generation and Bug Detection for Finite-State Machines. *Arxiv* (2024).

[6] Zhenni Bi, Kai Han, Chuanjian Liu, Yehui Tang, and Yunhe Wang. 2025. Forest-of-Thought: Scaling Test-Time Compute for Enhancing LLM Reasoning. *ICML* (2025).

[7] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large Language Monkeys: Scaling Inference Compute with Repeated Sampling. *Arxiv* (2024).

[8] Paul E. Calzada, Zahin Ibnat, Tanvir Rahman, Kamal Kandula, Danyu Lu, Sujan Kumar Saha, Farimah Farahmandi, and Mark Tehranipoor. 2025. VerilogDB: The Largest, Highest-Quality Dataset with a Preprocessing Framework for LLM-based RTL Generation. *Arxiv* (2025).

[9] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. *NeurIPS SysML Workshop* (2023).

[10] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. *CAV* (2023).

[11] Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, and Dahua Lin. 2024. OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection. *ICCAD* (2024).

[12] Chenhui Deng, Yun-Da Tsai, Guan-Ting Liu, Zhongzhi Yu, and Haoxing Ren. 2025. ScaleRTL: Scaling LLMs with Reasoning Data and Test-Time Compute for Accurate RTL Code Generation. *MLCAD* (2025).

[13] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, ..., and Zhen Zhang. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* (2025).

[14] Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xiangxi Mo, Eric Tang, Sumanth Hegde, Kourosh Hakhamaneshi, Shishir G Patil, Matei Zaharia, Joseph E Gonzalez, and Ion Stoica. 2025. LLMs Can Easily Learn to Reason from Demonstrations Structure, not content, is what matters! *Arxiv* (2025).

[15] Yuchao Liao, Tosiron Adegbija, and Roman Lysecky. 2024. Are LLMs Any Good for High-Level Synthesis? *ICCAD* (2024).

[16] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. *ICCAD* (2023).

[17] Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. 2025. CraftRTL: High-quality Synthetic Data Generation for Verilog Code Models with Correct-by-Construction Non-Textual Representations and Targeted Code Repair. *ICLR* (2025).

[18] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. *LAD Workshop* (2024).

[19] Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 2024. OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided Design RTL Generation. *ICCAD* (2024).

[20] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. *ASP-DAC* (2024).

[21] Ruiyang Ma, Yuxin Yang, Ziqian Liu, Jiaxi Zhang, Min Li, Junhua Huang, and Guojie Luo. 2024. VerilogReader: LLM-Aided Hardware Test Generation. *LAD Workshop* (2024).

[22] Daniel Mendoza, Christopher Hahn, and Caroline Trippel. 2024. Translating Natural Language to Temporal Logics with Large Language Models and Model Checkers. *FMCAD* (2024).

[23] Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *EMNLP* (2025).

[24] Bardia Nadimi and Hao Zheng. 2024. A Multi-Expert Large Language Model Architecture for Verilog Code Generation. *LAD Workshop* (2024).

[25] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. 2024. BetterV: Controlled Verilog Generation with Discriminative Guidance. *ICML* (2024).

[26] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2025. Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation. *TODAES* (2025).

[27] Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. 2025. Comprehensive Verilog Design Problems: A Next-Generation Benchmark Dataset for Evaluating Large Language Models and Agents on RTL Design and Verification. *Arxiv* (2025).

[28] Suresh Purini, Siddhant Garg, Mudit Gaur, Sankalp Bhat, Sohan Mupparapu, and Arun Ravindran. 2025. ArchXBench: A Complex Digital Systems Benchmark Suite for LLM Driven RTL Synthesis. *MLCAD* (2025).

[29] Ruidi Qiu, Grace Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. 2025. CorrectBench: Automatic Testbench Generation with Functional Self-Correction using LLMs for HDL Design. *DATE* (2025).

[30] Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. 2024. AutoBench: Automatic Testbench Generation and Evaluation Using LLMs for HDL Design. *MLCAD* (2024).

[31] Arun Ravindran, Aditya Patra, Vahid Babaey, and Suresh Purini. 2025. Survey and Benchmarking of Large Language Models for RTL Code Generation: Techniques and Open Challenges. *Preprints* (2025).

[32] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling LLM Test-Time Compute Optimally Can be More Effective than Scaling Parameters for Reasoning. *ICLR* (2025).

[33] Chuyue Sun, Christopher Hahn, and Caroline Trippel. 2023. Towards Improving Verification Productivity with Circuit-Aware Translation of Natural Language to SystemVerilog Assertions. *CAV Workshop* (2023).

[34] Kimia Tasnia, Alexander Garcia, Tasnuva Farheen, and Sazadur Rahman. 2025. VeriOpt: PPA-Aware High-Quality Verilog Generation via Multi-Role LLMs. *Arxiv* (2025).

[35] Kimia Tasnia and Sazadur Rahman. 2025. OPL4GPT: An Application Space Exploration of Optimal Programming Language for Hardware Design by LLM. *ASP-DAC* (2025).

[36] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. *DATE* (2023).

[37] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. VeriGen: A Large Language Model for Verilog Code Generation. *TODAES* (2024).

[38] Ning Wang, Bingkun Yao, Jie Zhou, Xi Wang, Zhe Jiang, and Nan Guan. 2025. Large Language Model for Verilog Generation with Code-Structure-Guided Reinforcement Learning. *ICLAD* (2025).

[39] Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations. *ACL* (2024).

[40] Weiqin Wang, Yile Wang, and Hui Huang. 2025. Ranked Voting based Self-Consistency of Large Language Models. *ACL Findings* (2025).

[41] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. *ICLR* (2023).

[42] Yiting Wang, Guoheng Sun, Wanghao Ye, Gang Qu, and Ang Li. 2025. VeriReason: Reinforcement Learning with Testbench Feedback for Reasoning-Enhanced Verilog Generation. *Arxiv* (2025).

[43] Anjiang Wei, Huanmi Tan, Tarun Suresh, Daniel Mendoza, Thiago S. F. X. Teixeira, Ke Wang, Caroline Trippel, and Alex Aiken. 2025. VeriCoder: Enhancing LLM-Based RTL Code Generation through Functional Correctness Validation. *NeurIPS DL4C Workshop* (2025).

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning

in large language models. *NeurIPS* (2022).
[45] Yuheng Wu, Azalia Mirhoseini, and Thierry Tambe. 2025. On the Role of Temperature Sampling in Test-Time Scaling. *Arxiv* (2025).
[46] Yuheng Wu, Jianwen Xie, Denghui Zhang, and Zhaozhuo Xu. 2025. DEL-ToM: Inference-Time Scaling for Theory-of-Mind Reasoning via Dynamic Epistemic Logic. *EMNLP* (2025).
[47] Claire Xenia Wolf. 2020. Equivalence Checking with Yosys. *GitHub* (2020).
[48] Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. 2025. AssertLLM: Generating Hardware Verification Assertions from Design Specifications via Multi-LLMs. *ASP-DAC* (2025).
[49] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS* (2023).
[50] Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine Lin, Yong Liu, and Haoxing Ren. 2025. Spec2RTL-Agent: Automated Hardware Code Generation from Complex Specifications Using LLM Agent Systems. *Arxiv* (2025).
[51] Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. 2025. MAGE: A Multi-Agent Engine for Automated RTL Code Generation. *DAC* (2025).