

000 001 002 003 004 005 PLUTO: A BENCHMARK FOR EVALUATING EFFI- 006 CIENCY OF LLM-GENERATED HARDWARE CODE 007 008 009

010 **Anonymous authors**
011 Paper under double-blind review
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053

ABSTRACT

Large Language Models (LLMs) are increasingly used to automate hardware design tasks, including the generation of Verilog code. While early benchmarks focus primarily on functional correctness, efficient hardware design demands additional optimization for synthesis metrics such as area, delay, and power. Existing benchmarks fall short in evaluating these aspects comprehensively: they often lack optimized baselines or testbenches for verification. To address these gaps, we present Pluto, a benchmark and evaluation framework designed to assess the efficiency of LLM-generated Verilog designs. Pluto presents a comprehensive evaluation set of 114 problems with self-checking testbenches and multiple Pareto-optimal reference implementations. Experimental results show that state-of-the-art LLMs can achieve high functional correctness, reaching 78.3% at pass@1, but their synthesis efficiency still lags behind expert-crafted implementations, with area efficiency of 63.8%, delay efficiency of 65.9%, and power efficiency of 64.0% at eff@1. This highlights the need for efficiency-aware evaluation frameworks such as Pluto to drive progress in hardware-focused LLM research.

1 INTRODUCTION

Large Language Models (LLMs) are beginning to reshape hardware design by automating key steps in hardware design workflows, including Verilog code generation Thakur et al. (2023a;b); Liu et al. (2023a), optimization Yao et al. (2024); Guo & Zhao (2025), verification Qiu et al. (2024a), debugging Tsai et al. (2024), high-level synthesis Xiong et al. (2024), and post-synthesis metric estimation Abdelatty et al. (2025). While these advances highlight the potential of LLMs in hardware design, most research has focused on functional correctness of generated designs, with little attention to design quality metrics such as area, delay, and power.

In hardware design, the quality of Verilog code is not determined solely by functional correctness. Designs typically undergo logic synthesis, where Verilog code is mapped to gate-level implementations in a target technology. This process exposes critical efficiency metrics, such as silicon area, timing delay, and power consumption, that directly impact manufacturability and performance. Unlike software code, where correctness and execution speed often suffice, hardware code quality is inherently tied to these post-synthesis metrics.

In order to evaluate the functional correctness of LLM-generated Verilog code, several benchmarks have been proposed including VerilogEval Liu et al. (2023a) and RTLLM Lu et al. (2024). Recent efforts, including RTLRewriter Yao et al. (2024), ResBench Guo & Zhao (2025), GenBen Wan et al. (2025), and TuRTLe Garcia-Gasulla et al. (2025), have begun to evaluate quality of LLM-generated hardware code in terms of post-synthesis metrics. However, these benchmarks face key limitations:

- **Absence of Optimal Ground Truth Solutions** True efficiency should be measured against implementations that are explicitly optimized for specific objectives such as silicon area, delay, or power consumption. Prior studies rely on canonical solutions from VerilogEval and RTLLM as reference solutions. Our analysis shows that these solutions are not the most optimal in terms of post-synthesis metrics.
- **Lack of Clock Latency Agnostic Testbenches** Many common optimization patterns, such as register pipelining, resource sharing, or FSM restructuring, introduce variations in clock-cycle latency between the optimized and unoptimized designs. To support fair evaluation,

054 testbenches must be self-checking and tolerant of different latency requirements. Existing
 055 benchmarks, however, assume identical latency between the reference model and the design
 056 under test, making them unsuitable for efficiency benchmarking.
 057

058 In order to address these limitations, we introduce *Pluto*, the first benchmark designed to evaluate
 059 both *functional correctness* and *synthesis efficiency* of LLM-generated Verilog code. Our contribu-
 060 tions are as follows:
 061

- **Per-Metric Ground Truth Optimal Solutions.** We provide a suite of 114 problems where each is optimized for area, delay, and power separately, yielding Pareto-front optimal solutions. Our ground truth solutions are significantly more efficient than canonical solutions in RTLLM and VerilogEval.
- **Optimization-Aware Testbenches.** Each problem is accompanied by clock-cycle agnostic testbenches that accommodate varying latency requirements, ensuring robust evaluation of different optimization patterns.
- **Comprehensive Evaluation.** We adapt the $eff@k$ metric introduced in Qiu et al. (2024b) to measure the efficiency of hardware designs. Our extended metric is a three-dimensional vector that evaluates LLM-generated code across multiple objectives: area, delay and power.

075 2 RELATED WORK

077 **Software Code Benchmarks** Large Language Models (LLMs) have been extensively studied for
 078 code generation across both software and hardware domains, with most early benchmarks focusing
 079 primarily on functional correctness rather than efficiency. In software, works such as Mercury Du
 080 et al. (2024) and ENAMEL Qiu et al. (2024b) move beyond correctness to explicitly evaluate run-
 081 time efficiency of LLM-generated programs. The Mercury Du et al. (2024) benchmark contains
 082 LeetCode style problems. Each problem is accompanied by an expert-written solution that repre-
 083 sents the most optimal implementation in terms of run-time efficiency. ENAMEL Qiu et al. (2024b)
 084 also introduces a Python benchmark to evaluate the run-time efficiency of LLM-generated code.

085 **Hardware Code Benchmarks** In hardware design, early work on LLM-generated Verilog empha-
 086 sized functional correctness. VerilogEval Liu et al. (2023a) only evaluates whether the LLM gen-
 087 erated code passes the testbench check, while RTLLM Lu et al. (2024) additionally checks if the
 088 generated code is synthesizable. More recent efforts have shifted toward assessing and improv-
 089 ing the efficiency of LLM-generated designs, which can be categorized into two main categories:
 090 *Specifications-to-Efficient-Verilog* where the LLM is tasked with translating natural language
 091 instruction to optimized Verilog code directly, and *Unoptimized-Verilog-to-Optimized-Verilog*, where
 092 the LLM is tasked with rewriting an unoptimized Verilog code to optimized Verilog code.

093 In the *Specifications-to-Efficient-Verilog* formulation, the LLM is prompted with a natural language
 094 problem description and directly generates optimized Verilog. Benchmarks, such as GenBen Wan
 095 et al. (2025), TuRTLe Garcia-Gasulla et al. (2025), evaluate these generations in functional correct-
 096 ness, synthesizability, and post-synthesis metrics such as area, delay, and power. However, it relies
 097 on VerilogEval problems as ground truth. These reference designs are not necessarily optimized
 098 for power, performance, or area, and thus do not represent true Pareto-optimal solutions. ResBench
 099 Guo & Zhao (2025) also does not define any gold-standard or reference-optimal implementations,
 100 which makes it difficult to quantitatively assess how close the generated solutions are to ideal results.

101 The *Unoptimized-Verilog-to-Optimized-Verilog* setting provides the LLM with a functionally cor-
 102 rect but unoptimized Verilog implementation and asks it to produce a more efficient version. RTL-
 103 Rewriter Yao et al. (2024) enhances this with retrieval-augmented generation and feedback through
 104 the synthesis loop. However, RTLRewriter lacks associated testbenches, making it unsuitable for
 105 assessing the functional correctness of the generated code.

106 As summarized in Table 1, *Pluto* is the first benchmark to offer per-metric optimization, provid-
 107 ing separate expert-optimized reference designs for area, delay, and power. This enables targeted,
 108 metric-specific evaluation of LLMs, an aspect missing from prior benchmarks.

108
 109 Table 1: Comparison of prior software and hardware code generation benchmarks. *Pluto* addresses
 110 key limitations by enabling metric-specific optimization with three reference implementations per
 111 problem, each optimized for area, delay, or power.

Benchmark	Language	Functionality	Synthesizability	Efficiency	Per-Metric Optimisation	Tasks
HumanEval	Python	✓	—	✗	✗	164
Mercury	Python	✓	—	✓	✗	256
ENAMEL	Python	✓	—	✓	✗	142
VerilogEval	Verilog	✓	✗	✗	✗	156
RTLLM	Verilog	✓	✓	✗	✗	30
RTLRewriter	Verilog	✗	✓	✓	✗	95
ResBench	Verilog	✓	✓	✗	✗	56
GenBen	Verilog	✓	✓	✗	✗	300
TuRTLe	Verilog	✓	✓	✗	✗	223
CVDP	Verilog	✓	✓	✗	✗	783
Pluto (Ours)	Verilog	✓	✓	✓	✓	114

3 PLUTO BENCHMARK

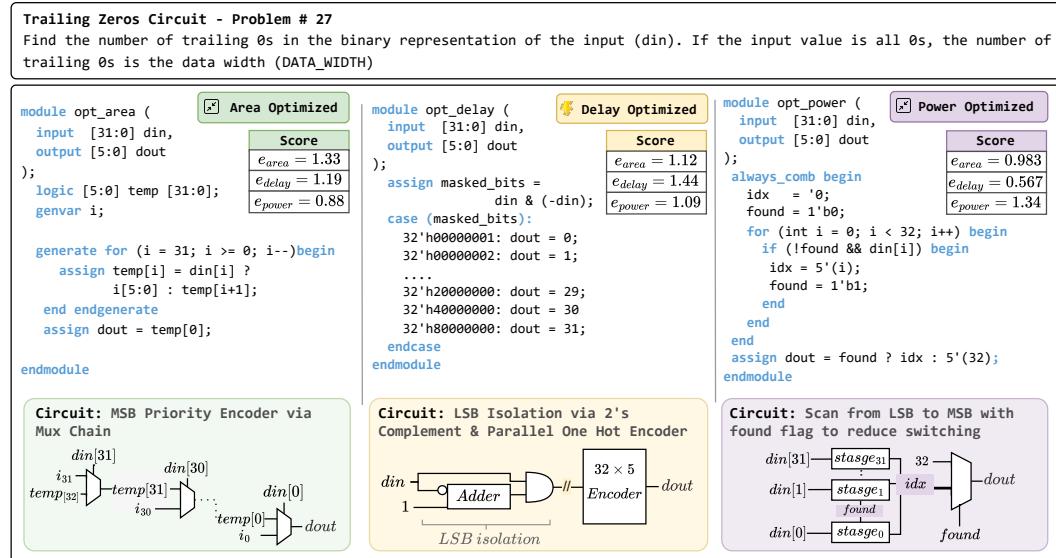


Figure 1: Overview of the *Pluto* benchmark on the trailing zeros detection task. We show three reference implementations optimized for different synthesis metrics compared to the unoptimized baseline: (left) area, using a mux-based priority encoder, reducing area by 33%; (center) delay, using an LSB isolation circuit with a parallel one-hot encoder, reducing delay by 44%; and (right) power, using an LSB-to-MSB scanning method with early termination, reducing total power by 34%. See Appendix. A.1 for unoptimized baseline and self-checking testbench.

3.1 DATA CONSTRUCTION

To enable a comprehensive evaluation of synthesis efficiency for LLM-generated hardware code, we construct the *Pluto* evaluation set, which contains a diverse collection of high-quality digital design problems spanning a broad range of difficulties. Specifically, we curated 114 problems from various publicly available sources, including open-source hardware projects, educational platforms such as ChipDev ChipDev (2025), a LeetCode-inspired platform for practicing Verilog coding, and prior benchmark suites such as RTLRewriter Yao et al. (2024), RTLLM Lu et al. (2024), and Ver-

ilogEval Liu et al. (2023b). Each problem is specified by a high-level description outlining the functional requirements, together with a baseline unoptimized Verilog implementation.

The problem set covers a wide spectrum of tasks in digital logic design, ranging from arithmetic units and control circuits to sequential state machines. To systematically capture variation in design complexity, we adopt ChipDev’s difficulty annotations and classify problems into three levels: easy, medium, and hard. These labels reflect the intrinsic challenge of translating the textual description into a correct Verilog implementation, thereby providing a principled way to distinguish between problems of different complexity.

Importantly, the resulting collection balances accessibility with challenge: many problems that appear straightforward can nonetheless expose substantial differences in synthesis efficiency depending on the optimization strategies applied. The diverse composition of easy, medium, and hard tasks therefore enables a nuanced assessment of an LLM’s ability to generate synthesis-efficient Verilog under varying constraints. In total, the 114 selected problems provide a representative and scalable testbed for benchmarking LLM-based Verilog efficiency.

Each problem instance in the *Pluto* benchmark includes the following components:

- **Prompt:** A natural language description of the hardware design task intended to guide the LLM-generation.
- **Module Header:** A fixed interface shared across all versions of the Verilog module to ensure consistency and comparability.
- **Unoptimized Verilog Code:** A baseline implementation used as the reference for testing.
- **Optimized Verilog Code:** Three distinct implementations with tradeoffs, each optimized by hand using design experts for a single metric: area, delay, or power.
- **Testbench:** A manually crafted, fully self-checking testbench that verifies functional equivalence between the unoptimized and any optimized design. These testbenches ensure full input space coverage and flag any mismatches during simulation. For sequential circuits, testbenches are clock-cycle agnostic, supporting latency differences introduced by optimizations such as pipelining or resource sharing.

All components in the evaluation set are manually developed. This ensures high quality and guarantees that the LLM under evaluation has not previously encountered any part of the dataset during training. In particular, the testbenches and optimized code serve as held-out ground truth references, providing an unbiased benchmark for assessing the efficiency and correctness of LLM-generated Verilog designs.

To illustrate the structure of problems in the *Pluto* evaluation set, Figure 1 presents the example of a trailing zeros detection circuit, categorized as an easy problem, along with its three metric-specific optimizations. As shown, each optimization achieves peak efficiency in its targeted metric, while performance in the remaining two metrics declines. This behavior emphasizes the inherent trade-offs across design objectives in hardware design and highlights the necessity of metric-specific optimization strategies.

3.2 OPTIMIZATION WORKFLOW

Each unoptimized design in the *Pluto* set is further refined through manual optimization by expert engineers to generate three distinct versions optimized separately for area, delay, and power. This workflow follows a systematic process that ensures both the correctness and the efficiency of the resulting designs. After applying metric-specific transformations, each optimized circuit is rigorously verified for functional correctness using Icarus Verilog Williams et al. (2002), supported by robust self-checking testbenches that guarantee equivalence with the unoptimized baseline. The optimized versions are then synthesized to confirm that improvements translate into measurable gains in area, timing, or power, thereby providing reliable performance baselines against which LLM-generated designs can be evaluated.

To understand how these efficiency gains are achieved, we visualize the optimization strategies applied across the dataset in appendix A.2. The strategies vary significantly depending on the target metric. For area, arithmetic optimizations and logic simplification are most commonly employed,

and FSM restructuring plays an important role in reducing redundant states and transitions. Delay improvements rely heavily on exploiting parallelism and restructuring control logic, often complemented by logic simplification and pipelining techniques that shorten the critical path. For power, it's reducing switching activity through register and logic optimizations, supported by techniques such as operand isolation, and clock gating to further suppress unnecessary toggling.

The distribution of strategies reveals that no single optimization technique dominates across all objectives. Instead, engineers select strategies tailored to the specific metric, reflecting the trade-offs inherent in digital design. As shown in Figure 2, this process results in consistent improvements across **all 114** designs, with average reductions of **19.19% (SD=18.99%)** in area, **21.96% (SD=20.99%)** in delay, and **22.55% (SD=21.65%)** in power. This highlights the importance of metric-specific approaches and provide a robust baseline for evaluating LLM-generated hardware code efficiency.

To further illustrate the impact of expert-driven optimization, Table 2 presents representative examples drawn from both RTLLM and VerilogEval. These case studies highlight how different strategies, such as arithmetic unit sharing, FSM encoding choices, and counter-based control logic, translate into concrete improvements across area, delay, and power. As shown, across both VerilogEval and RTLLM, expert-optimized designs consistently outperform baseline implementations. In particular, for RTLLM problems our expert-written solutions achieve average improvements of **18.75% (SD=14.55%)** in area, **22.75% (SD=20.99%)** in delay, and **20.43% (SD=21.65%)** in power compared to their canonical solutions. For VerilogEval problems, the improvements average **10.46% (SD=14.40%)** in area, **10.33% (SD=15.10%)** in delay, and **13.61% (SD=18.86%)** in power.

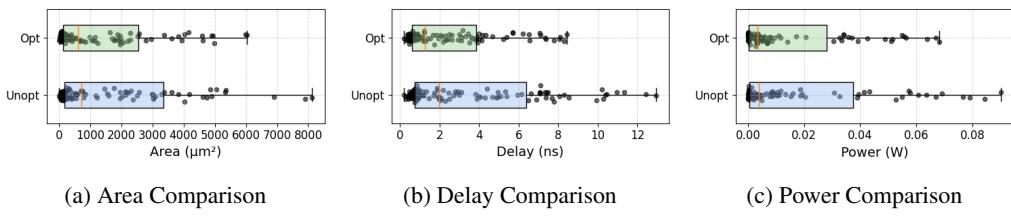


Figure 2: Distribution of area, delay, and power across **all 114 tasks** in *Pluto* before and after manual metric-specific optimizations.

Table 2: A sample of benchmark problems from *Pluto* dataset. Our expert-optimized solutions (area, delay, power) are significantly more efficient than the baseline benchmark implementations. See Appendix A.3 for full problem implementation.

ID	Source	Problem Description	Benchmark Solution	Expert Solution (Ours)
#60	RTLLM	ALU for a 32-bit MIPS-ISA CPU with operations like {ADD, SUB, AND, OR, XOR, SLT, shifts, LUI}.	ALU implementation with case statement and parameterized op-codes.	Area ↓ 26%: Shared adder for arithmetic, simplified flag logic, and operand reuse Delay ↓ 26%: Parallel datapaths with one-hot muxing Power ↓ 4%: Operand gating and early zeroing for large shifts to cut switching activity
#68	RTLLM	FSM detecting the sequence 10011 on a serial input stream with support for continuous and overlapping detection	States binary encoded, sequential next-state logic, and registered Mealy output	Area ↓ 32%: Casez-based transitions and direct output Delay ↓ 17%: One-hot state encoding with pre-decoded inputs and Moore-style output Power ↓ 23%: Compact binary encoding and casez-based transitions to cut toggling
#87	VerilogEval	Module controls a shift register enable signal, <i>shift_ena</i> asserted for 4 clock cycles on reset, then remain 0 until the next reset	Uses explicit states with next-state logic to drive <i>shift_ena</i>	Area ↓ 47%: Minimized register width (2-bit counter) and compact comparator logic Delay ↓ 37%: Wider counter (3 bits) to simplify comparison and reduce logic depth on the critical path Power ↓ 46%: Small counter reused
#104	VerilogEval	Conway's Game of Life with a 16×16 toroidal grid: each cell updates based on neighbor counts n (live if $n = 3$ or $n = 2 \& \text{ alive}$)	Straightforward RTL with per-cell neighbor recomputation and sequential summing	Area ↓ 19%: Shared neighbor computations across rows, bitwise rotations for wraparound, less duplicate summations Delay ↓ 37%: Parallel neighbor summation with carry-save adder tree and direct decode for 2 and 3 Power ↓ 36%: Reduced toggling via computation reuse

3.3 EFFICIENCY METRICS

We use the *pass@k* Liu et al. (2023a) for measuring the functional correctness of LLM-generated Verilog code. The *pass@k* metric, defined in appendix. A.4, measures the percentage of problems for which at least one of the top- k generated samples passes the self-checking testbench.

To evaluate the synthesis efficiency of functionally correct samples, we adapt the $eff@k$ introduced in Qiu et al. (2024b) to Verilog code. First, we introduce the efficiency score $e_{i,j}$, defined in Eq. 1, which quantifies how close an LLM-generated design is to optimal ground truth implementation. In this equation, $\hat{R}_{i,j}$ denotes the reported synthesis metric (e.g., area, delay, or power) for the j -th sample of problem i , $T_{i,j}$ denotes an upper bound beyond which the design is considered inefficient, and $R_{i,j}$ denotes the optimal (lowest) known reference value for that metric. A score of 1 indicates that the sample exactly matches the optimal reference, while a score of 0 indicates that it exceeds the acceptable threshold or is functionally incorrect.

$$e_{i,j} = \begin{cases} \frac{\max(0, T_{i,j} - \hat{R}_{i,j})}{T_{i,j} - R_{i,j}}, & \text{if } n_{i,j} \text{ is correct} \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

$$\text{eff}@k = \frac{1}{N} \sum_{i=1}^N \mathbb{E}_{J \subseteq \{1, \dots, n\}, |J|=k} \left[\max_{j \in J} e_{i,j} \right] \\ = \frac{1}{N} \sum_{i=1}^N \sum_{r=k}^n \frac{\binom{r-1}{k-1}}{\binom{n}{k}} e_{i,(r)}. \quad (2)$$

We then use the efficiency score $e_{i,j}$ for computing the $eff@k$, defined in Eq. 2 as the average of the best (i.e., highest) efficiency scores among the top- k functionally correct samples for each problem. We use the unbiased estimator introduced in Qiu et al. (2024b) for computing $eff@k$ which computes the expectation value over a random subset J of code samples with size K .

4 EVALUATION RESULTS

We evaluate 18 large language models (LLMs) using our *Pluto* benchmark, which includes proprietary LLMs, general-purpose foundation models, code-specialized models, and Verilog-tuned models. To comprehensively assess efficiency-aware generation, we consider the two problem formulations in *Pluto*: translating unoptimized Verilog code into optimized implementations, and generating optimized code directly from natural-language specifications. For the first problem formulation, only instruction-tuned models are evaluated, as code completion models generally reproduce the unoptimized code without meaningful improvements.

4.1 MAIN RESULTS

Table 3 (a) reports the *pass@k* and *eff@k* metrics for the first problem formulation, where the task is to re-write unoptimized Verilog into more efficient implementations. Several trends emerge. First, in terms of functional correctness (*pass@k*), domain-tuned models such as VeriThoughts-Inst-7B and RTLCoder-DeepSeek-V1 achieve performance comparable to much larger foundational models like DeepSeek-Chat, demonstrating the benefit of Verilog-specific training. However, in terms of synthesis efficiency (*eff@k*), all models exhibit a noticeable drop relative to their *pass@k* scores. This gap underscores a common limitation: while LLMs can generate functionally correct Verilog, they struggle to match the Pareto-efficient expert baselines across area, delay, and power.

Table 3 (b) reports the *pass@k* and *eff@k* metrics for the second problem formulation, where models are tasked with translating natural language specifications into optimized Verilog implementations. This task is more challenging, and as a result, both *pass@k* and *eff@k* scores are consistently lower across all models. Similar to the first formulation, all models also exhibit lower *eff@k* values compared to their corresponding *pass@k* scores, underscoring the persistent difficulty of generating designs that are not only functionally correct but also synthesis-efficient. However, the relative gap between *pass@k* and *eff@k* is smaller in this setting compared to the first formulation. This is because specification-to-RTL translation is substantially harder: models often struggle to produce functionally correct code in the first place, which suppresses both correctness and efficiency scores.

324 Table 3: Evaluation results using *Pluto* for two problem formulations: **P1: Unoptimized-Verilog-**
 325 **to-Optimized-Verilog** and **P2: Specifications-to-Optimized-Verilog**. *pass@k* measures functional
 326 correctness, while *eff@k* measures efficiency across area, delay, and power.

(a) P1: Unoptimized-Verilog-to-Optimized-Verilog

Model	pass@1	pass@5	pass@10	eff@1			eff@5			eff@10		
				Area	Delay	Power	Area	Delay	Power	Area	Delay	Power
GPT-3.5	0.325	0.517	0.594	0.271	0.296	0.282	0.462	0.491	0.450	0.540	0.568	0.520
GPT-4o-mini	0.506	0.705	0.751	0.469	0.476	0.467	0.662	0.677	0.639	0.714	0.744	0.687
DeepSeek-Chat	0.612	0.802	0.860	0.586	0.599	0.601	0.776	0.795	0.794	0.839	0.862	0.846
Llama-3.3-70B-Instruct	0.473	0.701	0.757	0.446	0.462	0.429	0.662	0.696	0.662	0.707	0.760	0.735
Llama-3.1-8B-Instruct	0.160	0.432	0.567	0.127	0.156	0.145	0.358	0.437	0.384	0.494	0.584	0.505
Mistral-7B-Instruct-v0.2	0.106	0.318	0.453	0.078	0.094	0.100	0.244	0.296	0.301	0.358	0.446	0.427
Mixtral-8x7B-v0.1	0.255	0.520	0.652	0.217	0.231	0.210	0.462	0.487	0.447	0.593	0.630	0.561
starcoder2-15b-instruct-v0.1	0.659	0.960	0.988	0.611	0.633	0.591	0.879	0.924	0.871	0.913	0.952	0.904
CodeLlama-70b-Instruct-hf	0.576	0.905	0.956	0.522	0.541	0.523	0.842	0.876	0.824	0.903	0.925	0.878
DeepSeek-Coder-33B	0.783	0.963	0.997	0.638	0.659	0.640	0.902	0.927	0.883	0.942	0.960	0.927
Qwen2.5-Coder-7B-Inst	0.479	0.785	0.866	0.438	0.452	0.419	0.710	0.759	0.741	0.785	0.848	0.833
yang-z/CodeV-QC-7B	0.231	0.416	0.506	0.211	0.208	0.187	0.381	0.390	0.361	0.455	0.491	0.442
RTLCoder-DeepSeek-V1	0.532	0.854	0.915	0.471	0.495	0.468	0.774	0.789	0.757	0.843	0.850	0.809
VeriThoughts-Inst.-7B	0.611	0.797	0.854	0.540	0.560	0.524	0.708	0.740	0.702	0.763	0.785	0.765

(b) P2: Specifications-to-Optimized-Verilog

Model	pass@1	pass@5	pass@10	eff@1			eff@5			eff@10		
				Area	Delay	Power	Area	Delay	Power	Area	Delay	Power
GPT-3.5	0.239	0.395	0.471	0.225	0.235	0.225	0.373	0.390	0.381	0.439	0.469	0.468
GPT-4o-mini	0.391	0.533	0.591	0.360	0.377	0.363	0.475	0.520	0.495	0.532	0.572	0.551
DeepSeek-Chat	0.557	0.688	0.719	0.545	0.552	0.528	0.689	0.680	0.651	0.726	0.710	0.684
Llama-3.3-70B-Instruct	0.363	0.541	0.594	0.348	0.345	0.342	0.515	0.533	0.510	0.564	0.602	0.557
Llama-3.1-8B-Instruct	0.087	0.224	0.301	0.075	0.073	0.081	0.189	0.184	0.204	0.270	0.251	0.279
Mistral-7B-Instruct-v0.2	0.030	0.108	0.164	0.024	0.015	0.024	0.078	0.067	0.088	0.112	0.117	0.134
Mixtral-8x7B-v0.1	0.082	0.176	0.222	0.082	0.068	0.079	0.172	0.131	0.163	0.202	0.166	0.211
starcoder2-15b-instruct-v0.1	0.243	0.454	0.512	0.226	0.249	0.220	0.429	0.466	0.409	0.489	0.525	0.458
CodeLlama-70b-Instruct-hf	0.212	0.446	0.532	0.202	0.207	0.194	0.418	0.440	0.437	0.498	0.535	0.524
DeepSeek-Coder-33B	0.257	0.429	0.482	0.231	0.254	0.246	0.387	0.424	0.421	0.446	0.490	0.468
Qwen2.5-Coder-7B-Inst	0.164	0.324	0.389	0.158	0.162	0.148	0.307	0.319	0.295	0.366	0.375	0.357
code-gen-verilog-16b	0.068	0.200	0.289	0.069	0.064	0.058	0.188	0.175	0.188	0.268	0.253	0.272
yang-z/CodeV-CL-7B	0.265	0.485	0.553	0.233	0.243	0.237	0.432	0.455	0.436	0.493	0.536	0.511
yang-z/CodeV-QC-7B	0.260	0.458	0.529	0.223	0.229	0.222	0.420	0.415	0.410	0.497	0.480	0.478
yang-z/CodeV-All-QC	0.175	0.317	0.374	0.150	0.162	0.144	0.297	0.304	0.256	0.368	0.379	0.284
RTLCoder-DeepSeek-V1	0.203	0.400	0.480	0.177	0.199	0.184	0.345	0.404	0.361	0.417	0.499	0.430
RTLCoder-Mistral	0.199	0.347	0.418	0.185	0.188	0.188	0.316	0.332	0.329	0.381	0.411	0.395
VeriThoughts-Inst.-7B	0.216	0.336	0.398	0.211	0.206	0.207	0.316	0.330	0.329	0.367	0.394	0.393

4.2 ABLATION STUDIES

In addition to the Verilog code writing style, post-synthesis metrics are also influenced by external factors such as the synthesis tool employed, the target technology library, and the optimization sequence executed by the tool. To understand the robustness of our proposed benchmark and isolate the impact of these factors, we present two ablation studies that evaluate efficiency trends in the *Pluto* benchmark across different synthesis tools, optimization strategies, and technology libraries.

4.2.1 SYNTHESIS TOOL AND TECHNOLOGY AGNOSTICISM

In this experiment, we repeated synthesis runs for the three optimized reference implementations in *Pluto*, as well as the unoptimized baseline, using two distinct synthesis tools: Yosys Wolf et al. (2013), an open-source framework, and Cadence Genus cad, a commercial synthesis tool. To further evaluate generalizability, we also targeted two technology libraries representing different fabrication nodes: the SkyWater 130nm library Google and a 65nm TSMC library tsm. We then computed the efficiency score for each tool and library configuration by comparing each optimized implementation against the corresponding unoptimized baseline across area, delay, and power metrics. As shown in Figure 3, efficiency scores remain consistent across all synthesis tool and technology combinations. This demonstrates that *Pluto*'s optimization patterns deliver consistent tradeoffs across different synthesis tools and technology libraries.

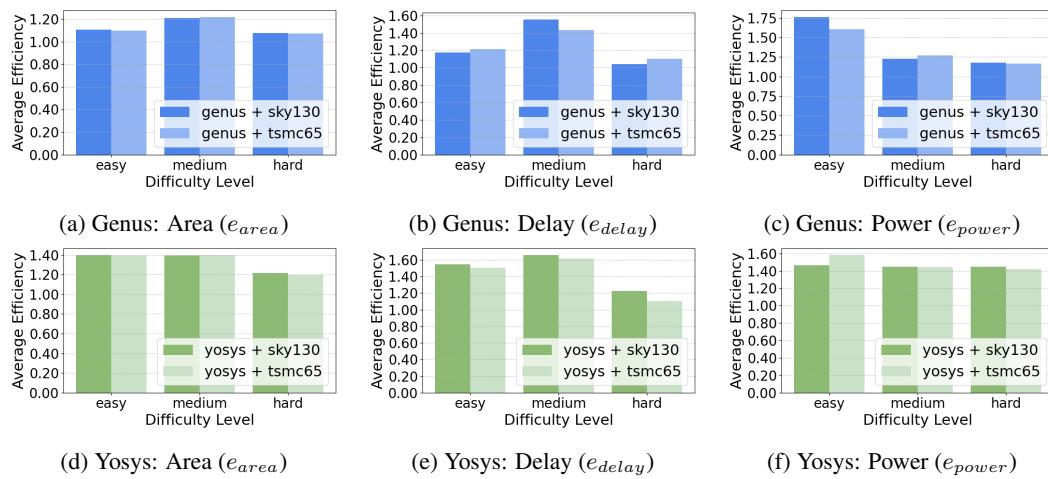


Figure 3: Efficiency scores for area, delay, and power across all benchmark problems, using both Cadence Genus and Yosys with different technology libraries. Results show consistent efficiency trends across synthesis tools and technologies.

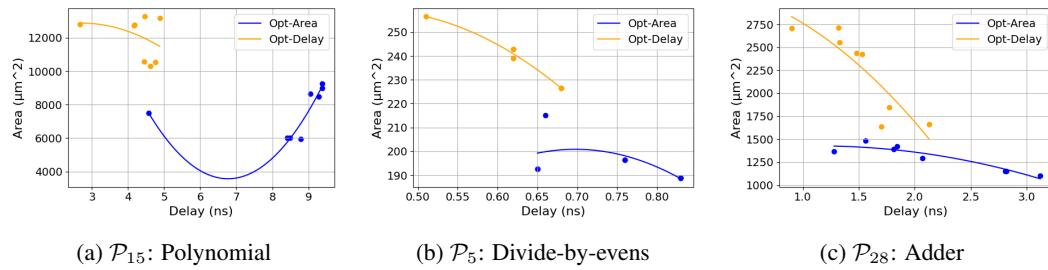


Figure 4: Area–delay tradeoffs of three problems in the *Pluto* benchmark under different synthesis strategies. Each strategy corresponds to a distinct sequence of ABC logic synthesis commands.

4.2.2 SYNTHESIS OPTIMIZATION STRATEGIES

We also examine how different synthesis optimization strategies influence the post-synthesis metrics of *Pluto*’s optimized implementations. Synthesis tools allow designers to specify optimization directives that steer the tool’s internal heuristics toward minimizing a particular metric while potentially sacrificing others. To study this effect, we synthesized selected problems from the *Pluto* benchmark under both area-optimized and delay-optimized optimization strategies. We used Yosys as our synthesis tool and targeted the SkyWater 130nm library. Within Yosys, logic optimization is carried out using the ABC framework Synthesis & Group (2024), which provides a collection of optimization heuristics that can be configured to emphasize different objectives such as area or delay minimization. Figure 4 illustrates the resulting Pareto fronts of area–delay trade-offs across three representative problems. As expected, delay-optimized code consistently achieves superior timing performance at the expense of larger area, whereas area-optimized code achieves lower area but incurs higher delays. These results confirm that synthesis settings primarily shift designs along the area–delay curve, while coding style remains the dominant factor, validating *Pluto*’s ability to capture design efficiency independent of synthesis optimization settings.

5 MULTI-OBJECTIVE OPTIMIZATION

Pluto also supports benchmarking LLMs for multi-objective optimization across area, delay, and power. For each problem, the individually optimized design variants represent distinct Pareto-optimal points. Together, they form a Pareto front that captures the trade-offs between these de-

432 Table 4: Multi-objective performance comparison across metric combinations using the P2 problem
 433 formulation (Specifications-to-Optimized-Verilog). Best metric set per model is bolded.
 434

435 436 437 Model	438 439 440 441 442 443 444 445 Area-Delay		438 439 440 441 442 443 444 445 Area-Power		438 439 440 441 442 443 444 445 Delay-Power		438 439 440 441 442 443 444 445 Area-Delay-Power	
	438 pass@10	438 eff@10	438 pass@10	438 eff@10	438 pass@10	438 eff@10	438 pass@10	438 eff@10
GPT-4o-mini	0.570	0.501	0.597	0.548	0.570	0.515	0.579	0.506
Llama3.3-70B-Instruct	0.649	0.615	0.605	0.577	0.640	0.600	0.605	0.567
Qwen2.5-Coder-7B-Instruct	0.395	0.358	0.395	0.364	0.421	0.393	0.430	0.402
Mixtral-8x7B	0.219	0.193	0.219	0.204	0.237	0.207	0.228	0.207
StarCoder2	0.500	0.465	0.526	0.499	0.526	0.502	0.465	0.440
CodeLlama-70B	0.526	0.490	0.553	0.529	0.535	0.499	0.535	0.512
DeepSeek-Coder-33B	0.474	0.444	0.518	0.480	0.491	0.463	0.465	0.439
yang-z/CodeV-QC-7B	0.526	0.474	0.526	0.472	0.544	0.488	0.509	0.472
RTLCoder-DeepSeek-V1	0.483	0.409	0.465	0.437	0.439	0.399	0.474	0.450
VeriThoughts-Instruct-7B	0.386	0.368	0.404	0.389	0.421	0.411	0.342	0.323

446
 447
 448 sign metrics, enabling the evaluation of LLM-generated code for efficiency across multiple objectives. To generalize the efficiency score to multiple metrics, we consider a task-specific metric set
 449 $\mathcal{M} \subseteq \{\text{area, delay, power}\}$, where the size of \mathcal{M} determines the dimensionality of the objective
 450 (e.g., two metrics for area-delay optimization or all three for full PPA evaluation). For each metric
 451 $m \in \mathcal{M}$, let $e_{i,j}^{(m)}$ denote the single-metric efficiency score (defined in Eq. 1). We then compute a
 452 multi-objective efficiency score as a weighted combination defined in Eq. 3. Finally, we obtain the
 453 multi-objective eff@k by substituting $e_{i,j}^{\text{multi}}$ for $e_{i,j}$ in Eq. 2.
 454

$$e_{i,j}^{\text{multi}} = \sum_{m \in \mathcal{M}} w_m e_{i,j}^{(m)}, \quad \sum_{m \in \mathcal{M}} w_m = 1, \quad w_m \geq 0. \quad (3)$$

455
 456 Table 4 shows the performance of different LLMs on multi-objective optimization across varying
 457 metric combinations. For each metric set \mathcal{M} , we report the multi-objective eff@k scores with equal
 458 weighting ($w_m=1/|\mathcal{M}|$) across all metrics in the set. The results show that, unlike single-metric
 459 evaluation, there is no universally easiest metric combination. For example, GPT-4o-mini peaks
 460 on Area-Power, while the Llama3.3-70B-Instruct model achieves its highest score on Area-Delay.
 461 Several mid-sized models, such as Mixtral and StarCoder2, achieve their best performance on
 462 Delay-Power. Across most models, the lowest scores appear on the Area-Delay or Area-Delay-Power
 463 combinations, reflecting the inherently competing nature of these metrics. Overall, these results
 464 indicate that multi-objective PPA optimization remains a challenging problem, and different LLMs
 465 exhibit different strengths depending on the metric set.
 466

467 6 FAILURE ANALYSIS AND INSIGHTS

468
 469 While LLMs reliably produce functionally correct Verilog, their ability to optimize is uneven across
 470 metrics. Area optimization is comparatively tractable, since it often reduces to logic simplification
 471 or FSM re-encoding. These area optimizations represent common, syntactic, pattern-like edits
 472 that appear frequently in code corpora and programmer-annotated examples, making them more
 473 learnable for LLMs. By contrast, delay requires identifying and shortening the critical path, and
 474 power depends on subtle factors like switching activity and memory usage. Crucially, area transforms
 475 are often local (e.g., simplifying a logic expression), while power and delay optimization
 476 typically requires global reasoning across the entire design (e.g., pipeline balancing, critical path
 477 restructuring). Current LLMs, especially smaller ones, struggle with these larger-scope transforma-
 478 tions. This difficulty is reflected in our quadrant analysis (Figure 5a and 8), where many delay- and
 479 power-optimized designs remain correct but fail to improve efficiency, whereas area optimizations
 480 succeed more often. Moreover, analysis of optimization strategies (Figure 5b) shows that the hard-
 481 est transformations are register optimizations for delay, followed by resource sharing for power, and
 482 sequential restructuring for delay. In contrast, strategies tied to area are easier, aligning with our
 483 observation that area is the most accessible metric for LLMs.
 484

485 Model scale and specialization strongly influence outcomes. Larger models (15B, 33B, 70B) cap-
 486 ture richer patterns and propose alternative architectures, showing stronger generalization across

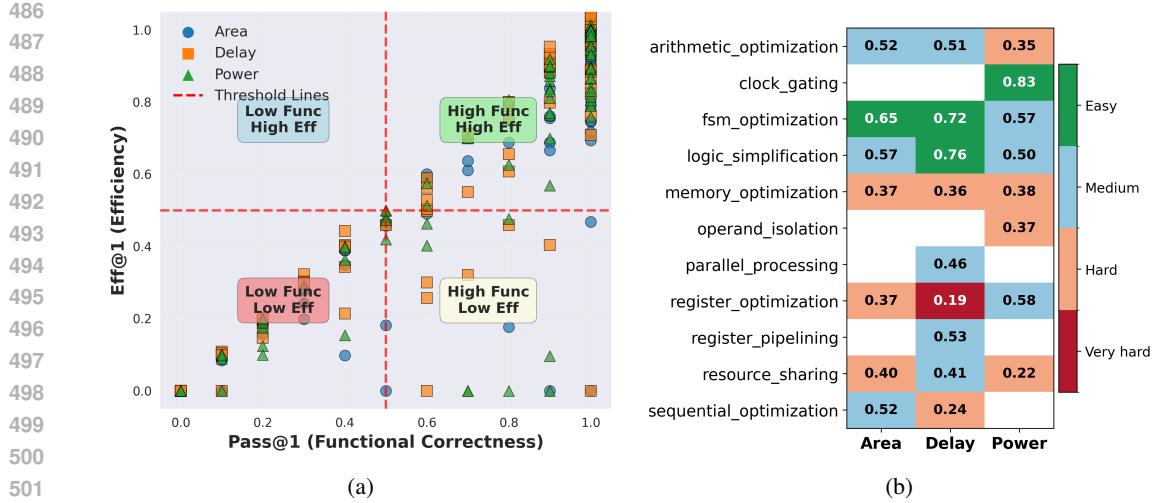


Figure 5: Failure mode analysis of optimization outcomes. (a) Quadrant plot showing the correlation between functional correctness ($Pass@1$) and synthesis efficiency ($Eff@1$) across area, delay, and power objectives. (b) Heatmap of optimization strategy difficulty across different optimization objectives area, delay, and power.

most optimization tactics. For example, larger code models perform more consistently on FSM optimization compared to smaller models, though they still struggle with register optimization and resource sharing. Models with explicit reasoning traces (e.g., DeepSeek, VeriThoughts) better decompose transformations and achieve stronger optimizations. Interestingly, register optimization lags behind other strategies across all DeepSeek variants (DeepSeek-Chat, DeepSeek-Coder, RTLCoder-DeepSeek-V1), suggesting this tactic requires capabilities beyond what current architectures capture, regardless of scale or specialization. Domain-tuned models outperform code models, which in turn outperform general-purpose LLMs, showing the value of Verilog-specific pretraining. Notably, generalist chat models show weaker performance on power-optimization techniques like clock gating and operand isolation, which are less prevalent in general training datasets.

Finally, a fundamental limitation is that Verilog training data lacks efficiency labels. LLMs therefore default to surface-level pattern matching rather than structural reasoning, and without feedback or synthesis-in-the-loop, they cannot tell whether changes reduce gate count or lengthen the critical path. Completion-style models exacerbate this issue, often rephrasing the baseline instead of innovating, whereas instruction-tuned models attempt more substantive edits. These findings suggest that true progress will require efficiency-focused benchmarks such as Pluto to guide future advances.

7 CONCLUSION

In this paper, we introduced *Pluto*, a comprehensive benchmark designed to evaluate the synthesis efficiency of LLM-generated Verilog code. *Pluto* provides an evaluation set of 114 hardware design problems, each accompanied by three reference optimized implementations (targeting area, delay, and power), an unoptimized baseline, a self-checking testbench, and a natural language description. Experimental results show that while LLMs can achieve high functional correctness, reaching up to 78.3% at $pass@1$, their synthesis efficiency remains limited: area efficiency of 63.8%, delay efficiency of 65.9%, and power efficiency of 64.0% at $eff@1$ compared to expert-crafted designs. These findings highlight the importance of efficiency-aware benchmarks beyond correctness alone and highlights the current limitations of LLMs in hardware optimization.

540 REFERENCES
541

542 Cadence Genus Synthesis Solution. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis.html.

544 TSMC 65nm Technology Library (Proprietary). Accessed under license from TSMC.

546 Manar Abdelatty, Jingxiao Ma, and Sherief Reda. Metrex: A benchmark for verilog code metric
547 reasoning using llms. In *Proceedings of the 30th Asia and South Pacific Design Automation
548 Conference*, pp. 995–1001, 2025.

549 ChipDev. ChipDev: Hardware Interview Prep & Verilog Practice. <https://chipdev.io>, 2025.
550 Accessed: 2025-01-01.

552 Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency
553 benchmark for code large language models. In *The Thirty-eight Conference on Neural Informa-
554 tion Processing Systems Datasets and Benchmarks Track*, 2024.

555 Dario Garcia-Gasulla, Gokcen Kestor, Emanuele Parisi, Miquel Albert’i-Binimelis, Cristian Gutier-
556 rez, Razine Moundir Ghorab, Orlando Montenegro, Bernat Homs, and Miquel Moreto. Turtle: A
557 unified evaluation of llms for rtl generation. *arXiv preprint arXiv:2504.01986*, 2025.

559 Google. Skywater-pdk. <https://github.com/google/skywater-pdk>. Accessed: 2025-
560 02-09.

561 Ce Guo and Tong Zhao. Resbench: Benchmarking llm-generated fpga designs with resource aware-
562 ness. *arXiv preprint arXiv:2503.08823*, 2025.

564 Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Invited paper: Verilogeval:
565 Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International
566 Conference on Computer Aided Design (ICCAD)*, pp. 1–8, 2023a. doi: 10.1109/ICCAD57390.
567 2023.10323812.

568 Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large
569 language models for verilog code generation. In *2023 IEEE/ACM International Conference on
570 Computer Aided Design (ICCAD)*, pp. 1–8. IEEE, 2023b.

572 Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl
573 generation with large language model. In *2024 29th Asia and South Pacific Design Automation
574 Conference (ASP-DAC)*, pp. 722–727. IEEE, 2024.

575 Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. Autobench: Au-
576 tomatic testbench generation and evaluation using llms for hdl design. In *Proceedings of the
577 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, MLCAD ’24, New
578 York, NY, USA, 2024a. Association for Computing Machinery. ISBN 9798400706998. doi:
579 10.1145/3670474.3685956. URL <https://doi.org/10.1145/3670474.3685956>.

581 Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How
582 efficient is llm-generated code? a rigorous & high-standard benchmark. *arXiv preprint
583 arXiv:2406.06647*, 2024b.

584 Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential synthesis and veri-
585 fication. Technical report, University of California, Berkeley, 2024. URL <https://people.eecs.berkeley.edu/~alanmi/abc/>.

587 Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri,
588 Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated
589 verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibi-
590 tion (DATE)*, pp. 1–6. IEEE, 2023a.

592 Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh
593 Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *ACM
Transactions on Design Automation of Electronic Systems*, 2023b.

594 YunDa Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with
 595 large language models. In *IEEE/ACM Design Automation Conference (DAC'24)*, pp. 1–8, 2024.
 596 doi: 10.1109/ICCAD57390.2023.10323812.

597

598 Gwok-Waa Wan, Yubo Wang, SamZaak Wong, Jingyi Zhang, Mengnv Xing, Zhe Jiang, Nan Guan,
 599 Ying Wang, Ning Xu, Qiang Xu, and Xi Wang. Genben: A generative benchmark for LLM-aided
 600 design. In *Proceedings of the International Conference on Learning Representations (ICLR)*,
 601 2025. URL <https://openreview.net/forum?id=gtV04xcpFI>. Under review.

602 Stephen Williams et al. Icarus verilog: open-source verilog more than a year later. *Linux Journal*,
 603 2002.

604

605 Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys—a free verilog synthesis suite. In *21st
 606 Austrian Workshop on Microelectronics (Austrochip)*, volume 97, 2013.

607

608 Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. Hlspilot: Llm-based high-level synthesis.
 609 *arXiv preprint arXiv:2408.06810*, 2024.

610

611 Xufeng Yao, Yiwen Wang, Xing Li, Yingzhao Lian, Ran Chen, Lei Chen, Mingxuan Yuan, Hong
 612 Xu, and Bei Yu. Rtlrewriter: Methodologies for large models aided rtl code optimization. *arXiv
 613 preprint arXiv:2409.11414*, 2024.

614 A APPENDIX

615 A.1 UNOPTIMIZED CODE AND TESTBENCH FOR PROBLEM #17 (TRAILING ZEROS) IN 616 FIGURE 1

617 A.1.1 UNOPTIMIZED CODE

```
618
619 1 module unopt_model #(parameter
620   2   DATA_WIDTH = 32
621   3   )
622   4   input  [DATA_WIDTH-1:0] din,
623   5   output logic [${clog2(DATA_WIDTH)}:0] dout
624   6   );
625
626   7   logic [DATA_WIDTH-1:0] din_adj;
627   8   logic [${clog2(DATA_WIDTH)}:0] idx;
628
629   9   always_comb begin
630   10      idx = 0;
631   11      din_adj = din & (~din+1);
632   12      for (int i=0; i<DATA_WIDTH; i++) begin
633   13          idx += (din_adj[i]) ? i : 0;
634   14      end
635   15
636   16      assign dout = (din_adj == 0 ? DATA_WIDTH : din_adj == 1 ? 0 : idx);
637   17
638   18 endmodule
```

639 A.1.2 SELF-CHECKING TESTBENCH

```
640
641 1 `timescale 1 ps/1 ps
642
643 2 module tb();
644
645 3   reg clk = 0;
646 4   initial forever #5 clk = ~clk;
647
648 5   wire [5:0] dout_opt, dout_unopt;
649 6   reg [31:0] din;
650
651 7   integer errors = 0;
652 8   integer errortime = 0;
653 9   integer clocks = 0;
654 10  integer total_cycles = 200;
655
656 11  initial begin
657
658 12    $dumpvars(0, tb);
659
660 13    $display("Starting simulation with %d total cycles", total_cycles);
661
662 14    for (int i=0; i<total_cycles; i++) begin
663
664 15      din = $urandomrange(0, 32);
665
666 16      $display("Input: %b", din);
667
668 17      $display("Output (Optimized): %b", dout_opt);
669
670 18      $display("Output (Unoptimized): %b", dout_unopt);
671
672 19      if (dout_opt != dout_unopt) begin
673
674 20        errors++;
675
676 21        $display("Error at cycle %d: Output mismatch", i);
677
678 22        if (errortime == 0) begin
679
680 23          errortime = i;
681
682 24          $display("First error detected at cycle %d", errortime);
683
684 25        end
685
686 26      end
687
688 27    end
689
690 28    $display("Simulation completed after %d cycles.", i);
691
692 29    if (errors > 0) begin
693
694 30      $display("Total errors: %d", errors);
695
696 31      $display("First error occurred at cycle %d", errortime);
697
698 32    end
699
700 33  end
701
702 34 end
703
704 35
705 36
```

```

648      $dumpfile("wave.vcd");
649      $dumpvars(1, clk, din, dout_opt, dout_unopt);
650
651      // Initialize din to avoid X values
652      din = 0;
653
654      // Generate random values for din
655      repeat(total_cycles) @ (posedge clk) din = $random;
656  end
657
658  wire tb_match;
659  assign tb_match = (dout_opt === dout_unopt);
660
661  opt_model opt_model (
662      .din(din),
663      .dout(dout_opt)
664  );
665
666  unopt_model unopt_model (
667      .din(din),
668      .dout(dout_unopt)
669  );
670
671  always @ (posedge clk) begin
672      clocks = clocks + 1;
673      if (!tb_match) begin
674          if (errors == 0) errortime = $time;
675          errors = errors + 1;
676      end
677
678      // Print the signals for debugging
679      $display("Time=%0t|Cycle=%0d|din=%h|opt=%h|unopt=%h|match=%b",
680              $time, clocks, din, dout_opt, dout_unopt, tb_match);
681
682      if (clocks >= total_cycles) begin
683          $display("Simulation completed.");
684          $display("Total mismatches: %ld out of %ld samples", errors, clocks);
685          $display("Simulation finished at %0d ps", $time);
686          $finish;
687      end
688  end
689
690  initial begin
691      #1000000
692      $display("TIMEOUT");
693      $finish();
694  end
695
696  endmodule
697
698
699
700
701

```

A.2 OPTIMIZATION STRATEGIES

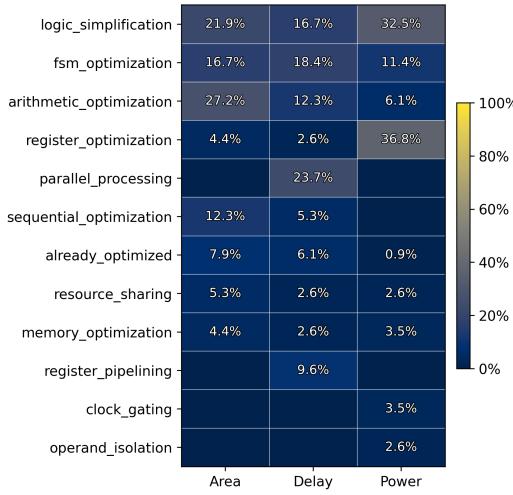


Figure 6: Optimization strategies employed for area, delay, and power improvements.

702
703

A.3 CODE OF EXAMPLE PROBLEMS IN TABLE 2

704
705
706
707
708
709
710

In the example problems shown in the appendix, the area- and power-optimized solutions coincided, as area-oriented designs also achieved the best power results, and vice versa. This overlap arises because common power-saving techniques, such as clock gating and operand isolation, were not applicable as some designs lacked a clock signal, while others did not include an enable signal. Consequently, explicit power-specific transformations could not be meaningfully applied. Moreover, in certain cases, power optimizations indirectly reduced area, further reinforcing the convergence of the two objectives into a single optimized implementation.

711
712

A.3.1 PROBLEM #60: RTLLM ALU

713
714
715
716
717
718
719
720

Problem description: Implement a 32-bit Arithmetic Logic Unit (ALU) for a MIPS-ISA CPU. The ALU takes two 32-bit operands (a and b) and a 6-bit control signal (aluc) that specifies which operation to perform. Based on this control signal, the ALU produces a 32-bit result (r) and several status outputs: zero indicates whether the result is zero, carry flags if a carry occurred, negative shows if the result is negative, overflow signals arithmetic overflow, and flag is used for set-less-than instructions (slt and sltu). The module supports arithmetic, logical, shift, and immediate load operations defined by specific opcodes (e.g., ADD, SUB, AND, OR, XOR, SLT, LUI).

Benchmark solution: ALU implementation with case statement and parameterized opcodes.

721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
53

```

1 module unopt_model(
2     input [31:0] a,
3     input [31:0] b,
4     input [5:0] aluc,
5     output [31:0] r,
6     output zero,
7     output carry,
8     output negative,
9     output overflow,
10    output flag
11 );
12
13 parameter ADD = 6'b100000;
14 parameter ADDU = 6'b100001;
15 parameter SUB = 6'b100010;
16 parameter SUBU = 6'b100011;
17 parameter AND = 6'b100100;
18 parameter OR = 6'b100101;
19 parameter XOR = 6'b100110;
20 parameter NOR = 6'b100111;
21 parameter SLT = 6'b101010;
22 parameter SLTU = 6'b101011;
23 parameter SLL = 6'b000000;
24 parameter SRL = 6'b000010;
25 parameter SRA = 6'b000011;
26 parameter SLLV = 6'b000100;
27 parameter SRLV = 6'b000110;
28 parameter SRAV = 6'b000111;
29 parameter JR = 6'b001000;
30 parameter LUI = 6'b001111;
31
32 wire signed [31:0] a_signed;
33 wire signed [31:0] b_signed;
34
35 reg [32:0] res;
36
37 assign a_signed = a;
38 assign b_signed = b;
39 assign r = res[31:0];
40
41 assign flag = (aluc == SLT || aluc == SLTU) ? ((aluc == SLT) ? (a_signed < b_signed) : (a < b)) : 1'bz;
42 assign zero = (res == 32'b0) ? 1'b1 : 1'b0;
43
44 always @ (a or b or aluc)
45 begin
46     case(aluc)
47         ADD: begin
48             res <= a_signed + b_signed;
49         end
50         ADDU: begin
51             res <= a + b;
52         end
53         SUB: begin

```

```

756      res <= a_signed - b_signed;
757 54      end
758 55      SUBU: begin
759 56          res <= a - b;
760 57      end
761 58      AND: begin
762 59          res <= a & b;
763 60      end
764 61      OR: begin
765 62          res <= a | b;
766 63      end
767 64      XOR: begin
768 65          res <= a ^ b;
769 66      end
770 67      NOR: begin
771 68          res <= ~ (a | b);
772 69      end
773 70      SLT: begin
774 71          res <= a_signed < b_signed ? 1 : 0;
775 72      end
776 73      SLTU: begin
777 74          res <= a < b ? 1 : 0;
778 75      end
779 76      SLL: begin
780 77          res <= b << a;
781 78      end
782 79      SRL: begin
783 80          res <= b >> a;
784 81      end
785 82      SRA: begin
786 83          res <= b_signed >>> a_signed;
787 84      end
788 85      SLLV: begin
789 86          res <= b << a[4:0];
790 87      end
791 88      SRLV: begin
792 89          res <= b >> a[4:0];
793 90      end
794 91      SRAV: begin
795 92          res <= b_signed >>> a_signed[4:0];
796 93      end
797 94      LUI: begin
798 95          res <= {a[15:0], 16'h0000};
799 96      end
800 97      default:
801 98      begin
802 99          res <= 32'bz;
803 100     end
804 101     endcase
805 102 end
806 103 endmodule
807 104

```

Our expert-written area and power optimized solution: Shared adder for arithmetic, simplified flag logic and operand reuse, leading to 26% area reduction. Operand gating and early zeroing for large shifts to cut switching activity, for 4% power reduction.

```

1      wire sub_mode    = (aluc==SUB) | (aluc==SUBU) | (aluc==SLT) | (aluc==SLTU);
2      wire [31:0] b_eff = sub_mode ? ~b : b;
3      wire      cin    = sub_mode;
4      wire [32:0] sum33 = {1'b0,a} + {1'b0,b_eff} + cin;
5
6      wire [31:0] add_res  = sum33[31:0];
7      wire      add_carry = sum33[32];
8
9      wire ovf = (a[31]^add_res[31]) & (b_eff[31]^add_res[31]);
10
11
12      wire signed_lt  = add_res[31] ^ ovf;
13      wire uns_lt     = ~add_carry;
14      wire [31:0] slt_res = {31'b0, signed_lt};
15      wire [31:0] sltu_res = {31'b0, uns_lt};
16
17      wire [31:0] and_res = a & b;
18      wire [31:0] or_res  = a | b;
19      wire [31:0] xor_res = a ^ b;
20      wire [31:0] nor_res = ~ (a | b);
21
22      wire [4:0] sa5    = a[4:0];
23      wire      any_hi  = |a[31:5];
24

```

```

810
811    wire [31:0] sll_full = any_hi ? 32'b0      : (b << sa5);
812    wire [31:0] srl_full = any_hi ? 32'b0      : (b >> sa5);
813    wire [31:0] sra_full = any_hi ? {32{b[31]}} : ($signed(b) >>> sa5);
814
815
816    wire [31:0] sllv_res = (b << a[4:0]);
817    wire [31:0] srlv_res = (b >> a[4:0]);
818    wire [31:0] srav_res = ($signed(b) >>> a[4:0]);
819
820
821    wire [31:0] lui_res = {a[15:0], 16'h0000};
822
823
824    reg [31:0] r_int;
825    always @* begin : result_mux
826        (* parallel_case, full_case *)
827        case (aluc)
828            ADD, ADDU: r_int = add_res;
829            SUB, SUBU: r_int = add_res;
830            AND: r_int = and_res;
831            OR: r_int = or_res;
832            XOR: r_int = xor_res;
833            NOR: r_int = nor_res;
834            SLT: r_int = slt_res;
835            SLTU: r_int = sltu_res;
836            SLL: r_int = sll_full;
837            SRL: r_int = srl_full;
838            SRA: r_int = sra_full;
839            SLLV: r_int = sllv_res;
840            SRLV: r_int = srav_res;
841            SRAV: r_int = srav_res;
842            LUI: r_int = lui_res;
843            JR: r_int = 32'bz;
844            default: r_int = 32'bz;
845        endcase
846    end
847
848    assign r    = r_int;
849    assign zero = ~(r_int);
850
851    assign carry    = 1'bz;
852    assign overflow = 1'bz;
853    assign negative = 1'bz;
854    assign flag     = (aluc==SLT) ? signed_lt :
855                    (aluc==SLTU) ? uns_lt    :
856                    1'bz;
857
858    endmodule
859
860

```

Our expert-written delay optimized solution: Parallel datapaths with one-hot muxing for shallow critical path, leading to 26% delay reduction.

```

861
862    wire signed [31:0] a_signed = a;
863    wire signed [31:0] b_signed = b;
864
865
866    wire [31:0] add_u  = a + b;
867    wire [31:0] sub_u  = a - b;
868    wire signed [31:0] add_s = a_signed + b_signed;
869    wire signed [31:0] sub_s = a_signed - b_signed;
870
871
872    wire [31:0] and_res = a & b;
873    wire [31:0] or_res  = a | b;
874    wire [31:0] xor_res = a ^ b;
875    wire [31:0] nor_res = ~(a | b);
876
877
878    wire slt_res  = (a_signed < b_signed);
879    wire sltu_res = (a < b);
880
881
882    wire [4:0] sham5 = a[4:0];
883
884
885    wire [31:0] sll_full  = (b << a);           // full 'a'
886    wire [31:0] srl_full  = (b >> a);           // full 'a'
887    wire [31:0] sra_full  = ($signed(b) >>> a_signed); // full signed 'a'
888    wire [31:0] sllv_res = (b << sham5);
889    wire [31:0] srlv_res = (b >> sham5);
890    wire [31:0] srav_res = ($signed(b) >>> sham5);
891
892
893    wire [31:0] lui_res = {a[15:0], 16'h0000};
894
895
896    wire sel_ADD  = (aluc==ADD);
897    wire sel_ADDU = (aluc==ADDU);
898    wire sel_SUB  = (aluc==SUB);
899    wire sel_SUBU = (aluc==SUBU);
900
901

```

```

864
865 33     wire sel_AND  = (aluc==AND);
866 34     wire sel_OR   = (aluc==OR);
867 35     wire sel_XOR  = (aluc==XOR);
868 36     wire sel_NOR  = (aluc==NOR);
869 37     wire sel_SLT   = (aluc==SLT);
870 38     wire sel_SLTU  = (aluc==SLTU);
871 39     wire sel_SLL   = (aluc==SLL);
872 40     wire sel_SRL   = (aluc==SRL);
873 41     wire sel_SRA   = (aluc==SRA);
874 42     wire sel_SLLV  = (aluc==SLLV);
875 43     wire sel_SRLV  = (aluc==SRLV);
876 44     wire sel_SRAV  = (aluc==SRAV);
877 45     wire sel_LUI   = (aluc==LUI);
878 46     wire sel_JR    = (aluc==JR);
879 47
880 48     wire any_sel = sel_ADD|sel_ADDU|sel_SUB|sel_SUBU|sel_AND|sel_OR|sel_XOR|sel_NOR|
881 49                     sel_SLT|sel_SLTU|sel_SLL|sel_SRL|sel_SRA|sel_SLLV|sel_SRLV|sel_SRAV|sel_LUI;
882 50
883 51     wire [31:0] r_known =
884 52         (sel_ADD ? add_s : 32'b0) |
885 53         (sel_ADDU ? add_u : 32'b0) |
886 54         (sel_SUB ? sub_s : 32'b0) |
887 55         (sel_SUBU ? sub_u : 32'b0) |
888 56         (sel_AND ? and_res: 32'b0) |
889 57         (sel_OR ? or_res : 32'b0) |
890 58         (sel_XOR ? xor_res: 32'b0) |
891 59         (sel_NOR ? nor_res: 32'b0) |
892 60         (sel_SLT ? {31'b0, slt_res} : 32'b0) |
893 61         (sel_SLTU ? {31'b0, sltu_res} : 32'b0) |
894 62         (sel_SLL ? sll_full : 32'b0) |
895 63         (sel_SRL ? srl_full : 32'b0) |
896 64         (sel_SRA ? sra_full : 32'b0) |
897 65         (sel_SLLV ? sllv_res : 32'b0) |
898 66         (sel_SRLV ? srlv_res : 32'b0) |
899 67         (sel_SRAV ? srav_res : 32'b0) |
900 68         (sel_LUI ? lui_res : 32'b0);
901 69
902 70     assign r = (any_sel && !sel_JR) ? r_known : 32'bz;
903 71
904 72     assign zero = (r == 32'b0) ? 1'b1 : 1'b0;
905 73
906 74     assign flag = (sel_SLT) ? slt_res :
907 75             (sel_SLTU) ? sltu_res :
908 76                 1'bz;
909 77
910 78     assign carry    = 1'bz;
911 79     assign negative = 1'bz;
912 80     assign overflow = 1'bz;
913 81 endmodule
914
915

```

A.3.2 PROBLEM #68: RTLLM FSM

Problem description: Implement a finite state machine (FSM) that detects the input sequence 10011 on a single-bit input stream. The module has three inputs: the serial input bit (IN), the clock (CLK), and a synchronous reset (RST). It produces one output, MATCH, which is asserted high when the specified sequence is recognized. The FSM supports continuous input and loop detection. When reset is active, the FSM initializes and MATCH is cleared to 0. The output MATCH is asserted during the cycle when the last 1 of the target sequence is received, and the design ensures that repeated or overlapping patterns (e.g., 100110011) correctly generate multiple match pulses.

Benchmark solution: States are binary-encoded, with sequential next-state logic in a Mealy FSM while output occupies a register.

```

909 1 module unopt_model (
910 2     input  wire IN,
911 3     input  wire CLK,
912 4     input  wire RST,
913 5     output wire MATCH
914 6 );
915 7
916 8     reg [2:0] ST_cr,ST_nt;
917 9
918 10    parameter s0 = 3'b000;
919 11    parameter s1 = 3'b001;
920 12    parameter s2 = 3'b010;
921 13    parameter s3 = 3'b011;
922 14    parameter s4 = 3'b100;

```

```

918
919 15 parameter s5 = 3'b101;
920 16
921 17 always@ (posedge CLK or posedge RST) begin
922 18     if(RST)
923 19         ST_cr <= s0;
924 20     else
925 21         ST_cr <= ST_nt;
926 22 end
927 23
928 24 always@(*) begin
929 25     case(ST_cr)
930 26         s0:begin
931 27             if (IN==0)
932 28                 ST_nt = s0;
933 29             else
934 30                 ST_nt = s1;
935 31         end
936 32
937 33         s1:begin
938 34             if (IN==0)
939 35                 ST_nt = s2;
940 36             else
941 37                 ST_nt = s1;
942 38         end
943 39
944 40         s2:begin
945 41             if (IN==0)
946 42                 ST_nt = s3;
947 43             else
948 44                 ST_nt = s1;
949 45         end
950 46
951 47         s3:begin
952 48             if (IN==0)
953 49                 ST_nt = s0;
954 50             else
955 51                 ST_nt = s4;
956 52         end
957 53
958 54         s4:begin
959 55             if (IN==0)
960 56                 ST_nt = s2;
961 57             else
962 58                 ST_nt = s5;
963 59         end
964 60
965 61         s5:begin
966 62             if (IN==0)
967 63                 ST_nt = s2;
968 64             else
969 65                 ST_nt = s1;
970 66         end
971 67
972 68     endcase
973 69 end
974 70
975 71 always@(*) begin
976 72     if(RST)
977 73         MATCH <= 0;
978 74     else if (ST_cr == s4 && IN == 1)
979 75         MATCH <= 1;
980 76     else
981 77         MATCH <= 0;
982 78 end
983 79
984 80 endmodule
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102
310
```

```

972      10      ST_cr <= ST_nt;
973      11      end
974      12
975      13      always @* begin
976      14          ST_nt = s0;
977      15          casez ({ST_cr, IN})
978      16              // s0: 0->s0, 1->s1
979      17              {s0,1'b0}: ST_nt = s0;
980      18              {s0,1'b1}: ST_nt = s1;
981      19
982      20              // s1: 0->s2, 1->s1
983      21              {s1,1'b0}: ST_nt = s2;
984      22              {s1,1'b1}: ST_nt = s1;
985      23
986      24              // s2: 0->s3, 1->s1
987      25              {s2,1'b0}: ST_nt = s3;
988      26              {s2,1'b1}: ST_nt = s1;
989      27
990      28              // s3: 0->s0, 1->s4
991      29              {s3,1'b0}: ST_nt = s0;
992      30              {s3,1'b1}: ST_nt = s4;
993      31
994      32              // s4: 0->s2, 1->s5
995      33              {s4,1'b0}: ST_nt = s2;
996      34              {s4,1'b1}: ST_nt = s5;
997      35
998      36              // s5: 0->s2, 1->s1
999      37              {s5,1'b0}: ST_nt = s2;
1000     38              {s5,1'b1}: ST_nt = s1;
1001     39
1002     40          default: ST_nt = s0;
1003     41      endcase
1004     42  end
1005     43
1006     44      assign MATCH = (ST_cr == s4) & IN;
1007     45
1008     46  endmodule
1009

```

Our expert-written delay optimized solution: One-hot state encoding with pre-decoded inputs and Moore-style output, leading to 23% delay reduction.

```

1      reg [5:0] S, S_next;
2
3      reg [5:0] S, S_next;
4
5      always @ (posedge CLK or posedge RST) begin
6          if (RST)
7              S <= 6'b000001;           // s0
8          else
9              S <= S_next;
10
11
12      wire in1 = IN;
13      wire in0 = ~IN;
14
15      always @* begin
16          S_next[0] = (S[0] & in0) | (S[3] & in0);           // -> s0
17          S_next[1] = (S[0] & in1) | (S[1] & in1) | (S[2] & in1) // -> s1
18                  | (S[5] & in1);                           // -> s2
19          S_next[2] = (S[1] & in0) | (S[4] & in0) | (S[5] & in0); // -> s3
20          S_next[3] = (S[2] & in0);                         // -> s4
21          S_next[4] = (S[3] & in1);                         // -> s4
22          S_next[5] = (S[4] & in1);                         // -> s5
23
24      end
25
26      always @ (posedge CLK or posedge RST) begin
27          if (RST)
28              MATCH <= 1'b0;
29          else
30              MATCH <= S[5];
31
32  endmodule

```

A.3.3 PROBLEM #87: VERILOGEVAL PROB095

Problem description: Implement a module that generates a control signal (shift_ena) for a shift register. The module has a clock input (clk), a synchronous active-high reset (reset), and a single

1026 output (shift_ena). The functionality requires that when the FSM is reset, the shift_ena signal is asserted high for exactly four consecutive clock cycles before being deasserted permanently. After this sequence, shift_ena remains low indefinitely until another reset occurs, at which point the behavior repeats. All sequential operations are triggered on the positive edge of the clock.

1027 **Benchmark solution:** Uses explicit states with next-state logic to drive shift_ena.

```

1031 1 module unopt_model (
1032 2   input  clk,
1033 3   input  reset,
1034 4   output reg shift_ena
1035 5 );
1036 6   parameter B0=0, B1=1, B2=2, B3=3, Done=4;
1037 7
1038 8   reg [2:0] state, next;
1039 9
104010   always @* begin
104111     case (state)
104212       B0:  next = B1;
104313       B1:  next = B2;
104414       B2:  next = B3;
104515       B3:  next = Done;
104616       Done: next = Done;
104717     default: next = B0;
104818   endcase
104919 end
105020
105121 endmodule

```

1052 **Our expert-written area and power optimized solution:** Minimized register width, using a 2-bit
1053 counter, and compact comparator logic, leading to 47% area reduction. Small counter reused which
1054 reduced toggling activity to minimize dynamic power, for 46% power reduction.

```

1056 1 reg [1:0] counter; // 2 bits are enough to count up to 4
1057 2
1058 3   always @ (posedge clk) begin
1059 4     if (reset) begin
1060 5       counter <= 2'b00; // Reset counter
1061 6       shift_ena <= 1'b1; // Enable on reset
1062 7     end else if (counter < 2'b11) begin
1063 8       counter <= counter + 1; // Increment counter
1064 9       shift_ena <= 1'b1; // Keep shift_ena high while counting
106510     end else begin
106611       shift_ena <= 1'b0; // Disable after 4 cycles
106712     end
106813   end
106914
107015 endmodule

```

1067 **Our expert-written delay optimized solution:** Wider counter, using 3 bits, to simplify comparison
1068 and reduce logic depth on the critical path, leading to 37% delay reduction.

```

1069 1 reg [2:0] count; // 3-bit counter to count 4 cycles
1070 2
1071 3   always @ (posedge clk) begin
1072 4     if (reset) begin
1073 5       count <= 3'b000; // Reset count
1074 6       shift_ena <= 1'b1; // Enable shift initially
1075 7     end else if (count < 3'b011) begin
1076 8       count <= count + 1; // Increment count
1077 9       shift_ena <= 1'b1; // Keep shift enabled
107810     end else begin
107911       shift_ena <= 1'b0; // Disable shift after 4 cycles
108012     end
108113   end
108214
108315 endmodule

```



```

1134
1135      end
1136  endgenerate
1137 // --- Precompute per-row horizontal neighbors (shared) -----
1138 wire [15:0] rol [15:0], ror [15:0];
1139 wire [15:0] sTrip [15:0], cTrip [15:0]; // for (L,C,R) of each row (0..3)
1140 wire [15:0] sPair [15:0], cPair [15:0]; // for (L,R) of each row (0..2)
1141
1142 genvar hr;
1143 generate
1144   for (hr = 0; hr < 16; hr = hr + 1) begin : HROW
1145     assign rol[hr] = roll(row[hr]);
1146     assign ror[hr] = rorl(row[hr]);
1147
1148   // Triplet = left + center + right (encoded as s + 2*c)
1149   wire [31:0] trip_pack = add3_vec(rol[hr], row[hr], ror[hr]);
1150   assign sTrip[hr] = trip_pack[15:0];
1151   assign cTrip[hr] = trip_pack[31:16];
1152
1153   // Pair = left + right
1154   assign sPair[hr] = rol[hr] ^ ror[hr];
1155   assign cPair[hr] = rol[hr] & ror[hr];
1156 end
1157 endgenerate
1158
1159 integer r;
1160 reg [255:0] nxt;
1161
1162 reg [3:0] rn, rp;
1163 reg [15:0] sT, cT, sM, cM, sB, cB;
1164 reg [15:0] ss, cS; // sum of (sT + sM + sB) as s + 2*c
1165 reg [15:0] U_is0, U_ge2, U_is1; // onehot(U) for U = cS + cT + cM + cB
1166 reg [15:0] is3, is2;
1167
1168 always @* begin
1169   nxt = '0;
1170   for (r = 0; r < 16; r = r + 1) begin
1171     rn = (r == 0) ? 4'd15 : r - 1;
1172     rp = (r == 15) ? 4'd0 : r + 1;
1173
1174     sT = sTrip[rn]; cT = cTrip[rn]; // top triplet from row r-1
1175     sM = sPair[r]; cM = cPair[r]; // middle pair (no center) from row r
1176     sB = sTrip[rp]; cB = cTrip[rp]; // bottom triplet from row r+1
1177
1178     {cS, ss} = add3_vec(sT, sM, sB);
1179
1180     U_is0 = ~(cS | cT | cM | cB);
1181     U_ge2 = ( (cS & cT) | (cS & cM) | (cS & cB)
1182           | (cT & cM) | (cT & cB) | (cM & cB) );
1183     U_is1 = ~(U_is0 | U_ge2);
1184
1185     is3 = ss & U_is1;
1186     is2 = ~ss & U_is1;
1187
1188     nxt[{r[3:0], 4'b0000} +: 16] = is3 | (row[r] & is2);
1189   end
1190 end
1191
1192 always @(posedge clk) begin
1193   if (load)
1194     q <= data;
1195   else
1196     q <= nxt;
1197 end
1198 endmodule

```

Our expert-written delay optimized solution: Parallel neighbor summation with carry-save adder tree and direct decode for 2 and 3, for shallow critical path, leading to 37% delay reduction.

```

1182 function automatic [7:0] idx(input [3:0] r, input [3:0] c);
1183   idx = {r, c};
1184 endfunction
1185
1186 function automatic [15:0] roll(input [15:0] x); roll = {x[14:0], x[15]}; endfunction
1187 function automatic [15:0] rorl(input [15:0] x); rorl = {x[0], x[15:1]}; endfunction
1188
1189 function automatic [31:0] add3_vec(input [15:0] a, input [15:0] b, input [15:0] c);
1190   add3_vec[15:0] = a ^ b ^ c; // s
1191   add3_vec[31:16] = (a & b) | (a & c) | (b & c); // c (>=2)
1192

```

```

118811      endfunction
118912
119013      integer r;
119114      reg [255:0] nxt;
119215
119316      reg [3:0] rn, rp;
119417      reg [15:0] ru, r0, rd;
119518      reg [15:0] ru_l, ru_c, ru_r;
119619      reg [15:0] r0_l, r0_c, r0_r;
119720      reg [15:0] rd_l, rd_c, rd_r;
119821
119922      reg [15:0] sT, cT, sM, cM, sB, cB, sS, cS;
120023      reg [15:0] U_is0, U_ge2, U_is1; // onehot decode for U = cS+cT+cM+cB
120124      reg [15:0] is3, is2; // neighbor count ==3 / ==2
120225
120326      always @* begin
120427          nxt = '0;
120528
120629          for (r = 0; r < 16; r = r + 1) begin
120730              rn = (r == 0) ? 4'd15 : r - 1;
120831              rp = (r == 15) ? 4'd0 : r + 1;
120932
121033              ru = q[{rn,4'b0000} +: 16];
121134              r0 = q[{r ,4'b0000} +: 16];
121235              rd = q[{rp,4'b0000} +: 16];
121336
121437              ru_l = roll(ru); ru_c = ru; ru_r = rorl(ru);
121538              r0_l = roll(r0); r0_c = r0; r0_r = rorl(r0);
121639              rd_l = roll(rd); rd_c = rd; rd_r = rorl(rd);
121740
121841              {cT, sT} = add3_vec(ru_l, ru_c, ru_r); // counts 0..3
121942              sM = (r0_l ^ r0_r); // pair: 0..2
122043              cM = (r0_l & r0_r);
122144              {cB, sB} = add3_vec(rd_l, rd_c, rd_r);
122245
122346              {cS, sS} = add3_vec(sT, sM, sB);
122447
122548              U_is0 = ~(cS | cT | cM | cB);
122649              U_ge2 = ( (cS & cT) | (cS & cM) | (cS & cB)
122750                  | (cT & cM) | (cT & cB) | (cM & cB) );
122851              U_is1 = ~(U_is0 | U_ge2);
122952
123053              is3 = sS & U_is1;
123154              is2 = ~sS & U_is1;
123255
123356              nxt[{r[3:0],4'b0000} +: 16] = is3 | (r0 & is2);
123457
123558          end
123659      end
123760
123861      always @ (posedge clk) begin
123962          if (load)
124063              q <= data;
124164          else
124265              q <= nxt;
124366      end
124467  endmodule

```

A.4 PASS@K DEFINITION

The pass@k metric, defined in equation 4, is used for measuring the functional correctness of LLM-generated code. Here, N is the total number of problems in the evaluation set. For a given problem i , we generate n_i samples and evaluate their functional correctness, obtaining c_i correct samples. The metric estimates the probability that at least one of the correct solution appears when drawing k samples.

$$\text{pass}@k = \mathbb{E}_{i=1}^N \left[1 - \frac{C(n_i - c_i, k)}{C(n_i, k)} \right] \quad (4)$$

where $C(n, k)$ denotes the binomial coefficient 'n choose k' and the term $1 - \frac{C(n_i - c_i, k)}{C(n_i, k)}$ represents the probability that at least one of the k samples is correct from the n_i total samples.

The metric averages this probability across all N problems to provide an unbiased estimate of the model's functional correctness when generating k samples per problem.

1242
 1243 Implementation Note: Following standard practice, we generate $n \geq k$ samples per problem (typ-
 1244 ically $n = 10$) and compute pass@k for various values of k (e.g., $k \in 1, 5, 10$) to obtain unbiased
 1245 estimates without requiring k separate evaluation runs.

1246 A.5 SAMPLING DIVERSITY
 1247

1248 In Figure 7, we show three LLM-generated Verilog modules of a parallel-in-serial-out shift register,
 1249 targeting area optimization, with normalized area efficiency values of 0.924, 0.963 and 1.0, respec-
 1250 tively. This diversity demonstrates why eff@k with $k > 1$ is valuable: generating multiple samples
 1251 increases the likelihood of obtaining high-quality implementations. The optimal sample (rightmost)
 1252 avoids unnecessary counters and state tracking present in the other solutions, reducing both area
 1253 and complexity, a correct solution that might not appear in a single generation attempt but becomes
 1254 accessible when drawing multiple samples.

```
1255
1256 module opt_model #(parameter DATA_WIDTH = 32) (
1257   input clk,
1258   input resetn,
1259   input [DATA_WIDTH-1:0] din,
1260   input din_en,
1261   output logic dout
1262 );
1263   reg [DATA_WIDTH-1:0] shift_reg;
1264   reg [$clog2(DATA_WIDTH)-1:0] bit_index;
1265   always @(posedge clk or negedge resetn) begin
1266     if (!resetn) begin
1267       shift_reg <= (DATA_WIDTH{1'b0});
1268       bit_index <= 0;
1269       dout <= 1'b0;
1270     end else if (din_en) begin
1271       shift_reg <= din;
1272       bit_index <= 0;
1273       dout <= din[0];
1274     end else if (bit_index < DATA_WIDTH-1) begin
1275       dout <= shift_reg[bit_index + 1];
1276       bit_index <= bit_index + 1;
1277     end else begin
1278       dout <= 1'b0;
1279     end
1280   end
1281 endmodule
1282
1283 module opt_model #(parameter DATA_WIDTH = 32) (
1284   input clk,
1285   input resetn,
1286   input [DATA_WIDTH-1:0] din,
1287   input din_en,
1288   output logic dout
1289 );
1290   reg [DATA_WIDTH-1:0] shift_reg;
1291   reg [$clog2(DATA_WIDTH)-1:0] bit_index;
1292   always @(posedge clk) begin
1293     if (din_en) begin
1294       shift_reg <= din;
1295       bit_index <= 0;
1296       dout <= din[0];
1297     end else if (bit_index < DATA_WIDTH-1) begin
1298       shift_reg <= shift_reg[bit_index + 1];
1299       bit_index <= bit_index + 1;
1300     end else begin
1301       dout <= 1'b0;
1302     end
1303   end
1304 endmodule
1305
1306 module opt_model #(parameter DATA_WIDTH = 32) (
1307   input clk,
1308   input resetn,
1309   input [DATA_WIDTH-1:0] din,
1310   input din_en,
1311   output logic dout
1312 );
1313   logic [DATA_WIDTH-1:0] temp;
1314   always @(posedge clk or negedge resetn) begin
1315     if (!resetn) begin
1316       temp <= 0;
1317     end else if (din_en) begin
1318       temp <= din;
1319     end else begin
1320       temp <= temp >> 1;
1321     end
1322   end
1323   assign dout = temp[0];
1324 endmodule
1325
```

(a) Top generation at $k = 1$ (b) Top generation at $k = 5$ (c) Top generation at $k = 10$

1270 Figure 7: Three area-optimized implementations of the piso_shift_register module (Prob-
 1271 lem #16) generated at $k \in \{1, 5, 10\}$. The circuit shifts the least significant bit of a multi-bit input
 1272 din to the single-bit output dout sequentially, starting when din_en goes high. All designs are
 1273 functionally correct but structurally diverse.

1274 A.6 FAILURE ANALYSIS
 1275

1276 In Figure. 8, we visualize the set of problems that have high pass@k and low eff@k to get better
 1277 insight on the set of problems that are hard to optimize.

1278 A.7 PROMPTING WITH OPTIMIZATION STRATEGIES
 1279

1280 To assess whether the relatively lower eff@k scores stem from missing optimization guidance, we
 1281 repeated the experiment using the first prompting strategy, where the LLM rewrites an unoptimized
 1282 Verilog module into a more efficient implementation, but this time we explicitly provided the opti-
 1283 mization goals and permissible strategies for each metric. The results, summarized in Table 5, show
 1284 that explicit guidance yields only marginal and model-dependent improvements. A few models
 1285 (e.g., Mixtral, StarCoder2-15B, CodeLlama-70B, and CodeV-QC-7B) show moderate gains across
 1286 several efficiency metrics, while others improve only in isolated cases (e.g., GPT-4o-mini improves
 1287 mainly in power, Qwen2.5-Coder-7B only in delay). However, for several strong models, includ-
 1288 ing DeepSeek-Chat and Llama-3.3-70B, efficiency scores worsen despite being given optimiza-
 1289 tion strategies. Overall, the gap between pass@k and eff@k remains substantial, indicating that the lim-
 1290 ited efficiency performance is not primarily caused by insufficient prompt specification, but rather
 1291 reflects the underlying difficulty of generating functionally correct and resource-efficient RTL si-
 1292 multaneously.

1296
1297
1298
1299
1300 Table 5: Efficiency results when using the first prompting strategy with explicit optimization goals
1301 and metric-specific optimization strategies. While a few models exhibit modest or selective improve-
1302 ments, most models show limited or no gains, and several regress. Overall, providing optimization
1303 guidance does not substantially narrow the gap between pass@k and eff@k.
1304

Model	pass@1			pass@5			pass@10			eff@1			eff@5			eff@10		
	Area	Delay	Power	Area	Delay	Power	Area	Delay	Power	Area	Delay	Power	Area	Delay	Power	Area	Delay	Power
GPT-4o-mini	0.471	0.667	0.725	0.435	0.440	0.423	0.622	0.637	0.642	0.696	0.693	0.716						
DeepSeek-Chat	0.502	0.728	0.795	0.487	0.496	0.485	0.717	0.719	0.709	0.795	0.789	0.774						
Llama-3.3-70B-Instruct	0.429	0.666	0.752	0.400	0.412	0.404	0.626	0.666	0.627	0.703	0.753	0.714						
Mixtral-8x7B-v0.1	0.271	0.562	0.696	0.232	0.255	0.244	0.491	0.528	0.508	0.607	0.648	0.632						
starcoder2-15b-instruct-v0.1	0.686	0.969	0.985	0.632	0.653	0.615	0.893	0.917	0.888	0.904	0.935	0.904						
CodeLlama-70b-Instruct-hf	0.599	0.904	0.944	0.538	0.559	0.547	0.834	0.857	0.846	0.877	0.905	0.895						
DeepSeek-Coder-33B	0.6088	0.9358	0.9766	0.5539	0.5706	0.5648	0.8745	0.8948	0.887	0.9270	0.9514	0.9191						
Qwen2.5-Coder-7B-Inst	0.443	0.756	0.839	0.399	0.421	0.389	0.675	0.735	0.692	0.745	0.834	0.785						
yang-z/CodeV-QC-7B	0.396	0.662	0.725	0.360	0.359	0.347	0.593	0.638	0.601	0.644	0.718	0.667						
RTLCoder-DeepSeek-V1	0.470	0.816	0.895	0.418	0.428	0.413	0.738	0.780	0.726	0.828	0.860	0.803						
VeriThoughts-Inst.-7B	0.545	0.770	0.836	0.475	0.508	0.492	0.673	0.715	0.699	0.748	0.774	0.758						

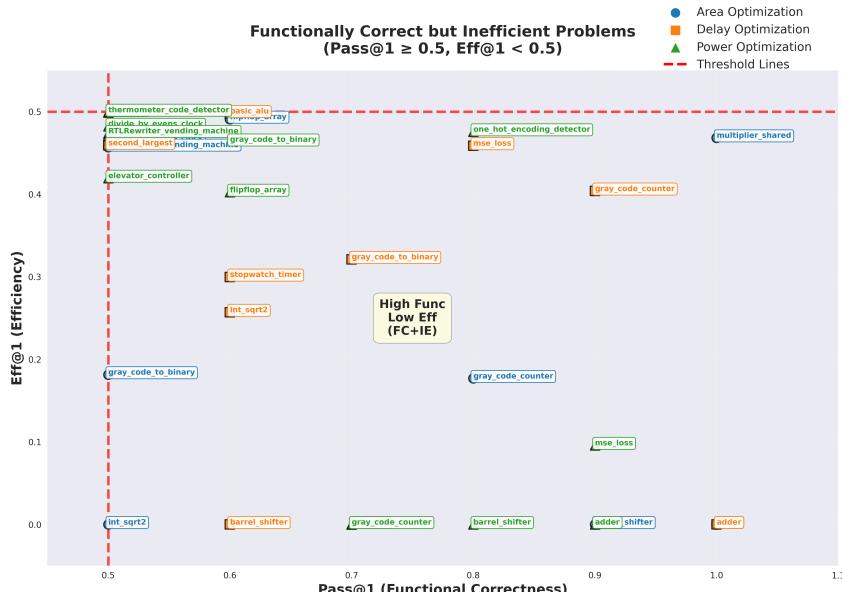


Figure 8: Quadrant plot showing the correlation between functional correctness (Pass@1) and synthesis efficiency (Eff@1) across area, delay, and power objectives.