

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Prompt-Based Inference of Design Constraints Using Generative AI

Madalena Sousa Silva



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Dr. João Canas Ferreira

October 9, 2025

Resumo

A crescente complexidade do projeto digital de ASIC intensificou a necessidade de diretivas precisas e eficientes, especificações críticas que orientam a verificação, a síntese e a implementação. Tradicionalmente, a definição destas diretivas é um processo moroso e propenso a erros, exigindo que os engenheiros traduzam diretrizes específicas de cada ferramenta em estruturas de comandos rigorosas. Esta dissertação propõe uma abordagem orientada por IA para simplificar esta tarefa, aproveitando LLMs potenciados com RAG e técnicas de engenharia de prompts para gerar diretivas precisas.

Foi desenvolvido um sistema protótipo que integra uma base de conhecimento composta por manuais de ferramentas numa pipeline RAG, capaz de recuperar dinamicamente a documentação relevante para auxiliar o LLM na geração de diretivas. A solução foi testada num conjunto de referência composto por cinco diretivas digitais, de forma a avaliar a sua robustez em diferentes cenários. Diversos LLMs de última geração foram avaliados sob diferentes estratégias de prompting e configurações de recuperação, explorando como a capacidade do modelo, os exemplos e a profundidade do contexto afetam o desempenho.

Para medir a eficácia de forma rigorosa, foi implementada uma framework de avaliação, dividida em quatro partes: uma análise quantitativa utilizando similaridade semântica e uma avaliação baseada num LLM, uma análise do tempo de inferência, uma inspeção com base em ferramentas, e uma revisão por especialistas em projeto ASIC. Os resultados mostram que o Llama 3.3 70B alcançou o melhor equilíbrio entre precisão, completude e tempo de inferência, gerando diretivas sistematicamente aceites pelas ferramentas.

Esta investigação demonstra que LLMs guiados por RAG têm a capacidade de reduzir significativamente o esforço manual na geração de diretivas, abrindo caminho à automação inteligente nos fluxos de projeto ASIC. Estabelece igualmente uma metodologia estruturada para avaliar diretivas geradas por IA e identifica os principais compromissos na seleção de modelos, na conceção de prompts e na configuração da recuperação. Por fim, este trabalho lança as bases para o desenvolvimento futuro de assistentes baseados em IA capazes de gerar, validar e refinar iterativamente diretivas em colaboração com ferramentas EDA de padrão industrial.

Abstract

The increasing complexity of ASIC digital design has intensified the demand for accurate and efficient design constraints, critical specifications that guide verification, synthesis and implementation. Traditionally, constraint definition is a labor-intensive and error-prone process, requiring engineers to translate tool-specific guidelines into precise command structures. This dissertation proposes an AI-driven approach to streamline this task, leveraging LLMs enhanced with RAG and prompting engineering to generate accurate design constraints.

A prototype system was developed, integrating a curated knowledge base of tool user guides into a RAG pipeline that dynamically retrieves relevant documentation for LLM-assisted constraint generation. The solution was tested on a benchmark suite of five designs to evaluate its robustness across varied scenarios. Multiple state-of-art LLMs were assessed under different prompting strategies and retrieval configurations, exploring how model capacity, examples, and context depth affect performance.

To rigorously measure effectiveness, a four-part evaluation framework was implemented: a quantitative analysis using semantic similarity and LLM-based correctness scoring, an inference time assessment, a tool-based inspection, and an expert review by ASIC design specialists. Results show that Llama 3.3 70B achieved the strongest balance between accuracy, completeness, and inference time, consistently generating constraints that were accepted by the tools.

This research demonstrates that RAG-guided LLMs have the potential to reduce manual effort in constraint generation, paving the way for intelligent automation in ASIC design flows. It also establishes a structured methodology for evaluating AI-generated constraints and identifies key trade-offs in model selection, prompt design, and retrieval configuration. At last, this work lays the foundation for future development of AI-powered designers capable of generating, validating, and iteratively refining constraints in collaboration with industry-standard EDA tools.

UN Sustainable Development Goals

As sustainability becomes an essential concern in technology development, it is critical to evaluate how engineering innovation aligns with the UN Sustainable Development Goals (SDGs). This section examines the alignment of this dissertation with the UN SDGs. While this dissertation introduces significant innovation and efficiency improvements, it also impacts consumption resources.

SDG 8 Promote inclusive and sustainable economic growth and employment for all.

SDG 9 Build resilient infrastructure, promote sustainable industrialization and foster innovation.

SDG 12 Ensure sustainable consumption and production patterns, which is key to sustain the livelihoods of current and future generations.

SDG	Target	Contribution	Performance Indicators and Metrics
8	8.2	Increases productivity of engineering teams by automating design constraining. Supports digital up skilling and enables faster time-to-market.	Reduction in engineering hours spent on constraint definition, increase in throughput of digital designs and employment trained in AI-assisted workflows.
9	9.4	Upgrades digital infrastructure by integrating AI into industrial design workflows, promoting sustainable and intelligent automation.	Number of constraint generation tasks automated, time reduction in ASIC development cycles and deployment of AI-accelerated tools in production workflows.
	9.5	Enhances R&D by embedding AI in digital system design, fostering innovation in semiconductor engineering.	Number of innovation projects supported and AI model iterations integrated into design infrastructure.
12	12.2	Reduces consumption of resources in IPs design processes by optimizing design iterations.	Lower energy costs associated with verification and synthesis reruns.
	12.5	Promotes efficiency in design workflows that contribute to waste prevention and sustainable engineering practices.	Decrease in silicon waste from unsuccessful fabrications rounds.

Acknowledgements

Quero expressar a minha mais profunda gratidão ao Prof. João Canas Ferreira, pelo seu apoio e orientação ao longo de todo este percurso. A sua disponibilidade, conhecimento e conselhos foram fundamentais para a concretização desta dissertação.

Agradeço também ao meu supervisor, Eng. Duarte Azevedo, por todo o apoio e, claro, um agradecimento especial ao Eng. Bernardo Costa, que dedicou incontáveis horas a ajudar-me a resolver todos os problemas técnicos que foram surgindo.

Estou igualmente grata à minha colega e amiga Patrícia, que esteve sempre presente, mesmo que à distância. A sua presença constante mostrou-me que as dificuldades partilhadas pesam metade, e as vitórias partilhadas sabem o dobro.

Ao meu namorado, Marco, agradeço com todo o meu coração. Obrigada por me apoiares nos dias bons e nos dias difíceis, por me fazeres rir quando mais precisava e por nunca deixares que eu esquecesse o meu valor, mesmo quando as dúvidas falavam mais alto.

Por fim, à minha família, que é o meu alicerce em tudo. Um agradecimento especial à minha mãe, o meu maior exemplo de força e inspiração. A sua resiliência e determinação são a bússola que sigo, e a sua fé em mim foi a força silenciosa que me sustentou em cada desafio desta dissertação.

Madalena Sousa Silva

Contents

1	Introduction	1
1.1	Background and Context	1
1.2	Motivation and Problem Statement	1
1.3	Research Objectives	2
1.4	Research Contributions	3
1.5	Research Questions	3
1.6	Dissertation Structure	4
2	ASIC digital design	5
2.1	Constraints in the ASIC design workflow	5
2.2	Multiple-clock designs	6
2.2.1	Setup and hold-time constraints	7
2.2.2	Metastability	8
3	Large Language Models	10
3.1	Terms and Definitions	10
3.2	Evolution of LLMs	11
3.3	Transformer Architecture and self-attention mechanism	12
3.4	Fine-tuning and Prompting techniques	13
3.4.1	Fine-tuning Strategies	13
3.4.2	Prompt Engineering	14
3.5	Trends in Leading Large Language Models	15
3.5.1	Advanced Reasoning and Thinking Systems	15
3.5.2	Optimization and Cost Efficiency	16
3.5.3	Lightweight and Portable Open-Source Models	16
3.6	Performance Benchmarks	16
3.7	Large Language Models in Electronic Design Automation	17
3.7.1	Current State of LLM Integration in EDA	17
3.7.2	Summary of Approaches and Future Directions	18
4	Retrieval-Augmented Generation	20
4.1	Introduction to RAG	20
4.2	Benefits and Applications of RAG	21
4.3	Optimization of RAG	22
4.4	Current challenges and Limitations of RAG	23

5	Proposed Solution and Implementation	24
5.1	Solution Overview	24
5.2	Benchmark Design Development	26
5.2.1	Design Selection and Motivation	26
5.2.2	Development Flow	27
5.2.3	Design Specifications	28
5.3	Implementation Decisions	29
5.3.1	Framework	29
5.3.2	Vector Database	30
5.3.3	Embedding Models	31
5.3.4	Language Models	32
5.4	Similarity and LLM-based Scores	35
5.5	RAG Implementation and Parametrization	36
5.5.1	RAG Implementation	36
5.5.2	RAG Parameters Variations	39
6	Results and Evaluation of RAG-based prototype	40
6.1	Quantitative Evaluation	40
6.2	Inference Time Analysis	46
6.3	Tool-based Inspection	48
6.4	Expert Review	50
7	Conclusions and Future Work	52
7.1	Research Answers	52
7.2	Main Contributions	53
7.3	Future Work	54
	References	55
A	Benchmark Designs	61
A.1	Reset synchronizer module	61
A.2	Asynchronous FIFO Design	62
A.2.1	Architecture Overview	62
A.2.2	High-Level Operation	62
A.2.3	Gray Code in FIFOs	63
A.2.4	Verilog Implementation	63
A.3	Synchronous FIFO Design	67
A.3.1	High-Level Operation	68
A.3.2	Verilog Implementation	68
A.4	Dual-Port RAM Design	69
A.4.1	Architecture Overview	70
A.4.2	High-Level Operation	70
A.4.3	Verilog Implementation	70
A.5	Single-Port RAM Design	73
A.5.1	Architecture Overview	73
A.5.2	High-Level Operation	73
A.5.3	Verilog Implementation	73
A.6	Clock Divider Design	75
A.6.1	Architecture Overview	75

A.6.2	High-Level Operation	75
A.6.3	Verilog Implementation	76
B	Design constraints	78
B.1	Clock constraints	78
B.2	Timing constraints	79
B.2.1	Disable Timing	81

List of Figures

2.1	Constraint-driven flow in ASIC design (from [4]).	6
2.2	Multiple clock system (from [4]).	7
2.3	Setup and hold-time (from [7]).	7
2.4	Timing diagram for a dual clock system (from [7]).	8
4.1	RAG flow (from [43]).	21
5.1	System’s pipeline.	25
5.2	Digital design flow followed for benchmark development.	28
5.3	Zero-shot CoT prompt designed for the system.	37
5.4	Few-shot CoT prompt designed for the system.	38
6.1	Box plot of similarity scores by configuration.	41
6.2	Box plot of correctness scores by configuration.	41
6.3	Box plot of similarity scores by prompt strategy.	42
6.4	Box plot of correctness scores by prompt strategy.	42
6.5	Box plot of similarity scores by model.	43
6.6	Box of for correctness scores by model.	43
6.7	Heatmap of final results.	44
6.8	Average Inference Time of configurations by model.	47
6.9	Average Inference Time of few-shot and zero-shot prompts by model.	47
B.1	Synchronous clock waveforms (from [76]).	78
B.2	Two-Phase Clocking Example for Setting Port Delays (from [76]).	80

List of Tables

5.1	Design specifications: target vs. maximum achievable frequencies.	29
5.2	Benchmark Scores of Llama 3.1 8B Instruct (from [67]).	32
5.3	Benchmark scores for Gemma 3 27B (from [68]).	33
5.4	Benchmark scores for Mistral-small 3.1 24B Instruct (from [69]).	33
5.5	Benchmark scores for Llama3.3-70B-Instruct (from [70]).	34
5.6	Benchmark scores for DeepSeek-R1 distill Llama 70B (from [71]).	34
5.7	Benchmark scores for Qwen2.5 Coder 14B and Qwen2.5 Coder 32B (from [72]).	35
6.1	Feedback of constraints generated for VC SpyGlass by Mistral model.	48
6.2	Feedback of constraints generated for VC SpyGlass by Llama 3.3 model.	48
6.3	Feedback of constraints generated for Design Compiler by Mistral model.	49
6.4	Feedback from experts for all designs.	50
A.1	3-bit mapping: Binary and Gray codes	63

List of Acronyms

AI	Artificial Intelligence
AIME	AI Model Evaluation
API	Application Protocol Interface
ASIC	Application-Specific Integrated Circuit
BERT	Bidirectional Encoder Representations from Transformers
BLEU	Bilingual Evaluation Understudy
CDC	Clock-Domain Crossing
CLM	Causal Language Models
CoT	Chain-of-Thought
DFT	Design For Testability
DRC	Design Rule Checks
EDA	Electronic Design Automation
FF	Flip-Flop
FIFO	First In, First Out
GPU	Graphics Processing Unit
GPQA	Graduate-Level Google-Proof Q&A
GPT	Generative Pre-Trained Transformer
HDL	Hardware Description Language
HNSW	Hierarchical Navigable Small World
ICL	In-Context Learning
IP	Intellectual Property
LLMs	Large Language Models
LMs	Language Models
LoCo	Long-Context
LoRA	Low-Rank Adaptation
LSTM	Long Short-Term Memory
ML	Machine Learning
MLM	Masked Language Model
MMLU	Massive Multitask Language Understanding
MoE	Mixture of Experts
MRR	Mean Reciprocal Rank
MTBF	Mean Time Between Failures
MTEB	Massive Text Embedding Benchmark
nDCG	Normalized Discounted Cumulative Gain
NLM	Neural Language Model
NLP	Natural Language Processing
OpenQA	Open-domain Question Answering
PDF	Portable Document Format
PEFT	Parameter-Efficient Fine-Tuning
PLM	Pre-trained Language Model
PPA	Power, Performance and Area
PVT	Process, Voltage, Temperature
QAT	Quantization-Aware Training
QLoRA	Quantized Low-Rank Adaptation

RAM	Random Access Memory
RAG	Retrieval-Augmented Generation
RDC	Reset-Domain Crossing
RLHF	Reinforcement Learning from Human Feedback
RNN	Recurrent Neural Network
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
RTL	Register Transfer Language
SFT	Supervised Fine-Tuning
STA	Static Timing Analysis
STEM	Science, Technology, Engineering and Mathematics
SWE-bench	Software Engineering Benchmark
TF-IDF	Term Frequency-Inverse Document Frequency
TPU	Tensor Processing Unit
VCS	Verilog Compiler Simulator

Chapter 1

Introduction

1.1 Background and Context

Application-Specific Integrated Circuit (ASIC) digital design is a keystone of modern electronics, enabling the creation of highly specialized and efficient circuits tailored to specific functional needs. The increasing complexity of these designs not only demands extensive resources for design and verification but also significantly extends the development cycle, emphasizing the critical need for tools and methodologies that can simplify the IC design flow.

In parallel, the rapid advancement of Artificial Intelligence (AI) and Large Language Models (LLMs) has revolutionized the ability to automate complex, knowledge-intensive tasks. Industries are increasingly adopting AI-driven workflows to enhance their productivity and reduce dependence on manual processes. In this evolving technological landscape, ASIC digital design stands out as a particularly promising domain for the deployment of these innovations, offering unprecedented capabilities to optimize and automatize several aspects of a digital design [1].

To ensure that the circuits meet stringent performance and functional standards, engineers rely on design constraints, parameters that guide the development process from verification to implementation and ultimately shape the final product. However, as ASIC designs grow in scale and complexity, there is an increasing recognition that traditional manual methods for defining these constraints could be further improved and optimized. This evolving context naturally opens the discussion on how emerging technologies can support, enhance, and ultimately reshape this essential aspect of ASIC digital design.

1.2 Motivation and Problem Statement

Following the advancements of digital systems, ASIC designs are becoming increasingly complex, demanding ever-higher standards in performance, power efficiency, and reliability. To meet these expectations, design constraints require precise formulation.

Engineers are often required to possess deep domain expertise, meticulously consult design manuals, and manually translate complex requirements into constraint files. This process is not

only time-consuming but also prone to human error, which can lead to delays in the design cycle and cause extra costs. Moreover, the absence of a standard methodology for defining constraints contributes to inconsistencies across projects and teams, creating confusion and potential quality issues.

Adding to the complexity, different tools in the ASIC design flow, ranging from early analysis to implementation, rely on different constraint formats. A standard format for design constraints across the different phases of the design process would speed ASIC design flow and guarantee greater quality in the consequent designs [2].

The motivation behind this dissertation is to address these inefficiencies by harnessing the power of AI, introducing an automated system for constraint generation. There is a clear need for a system capable of simplifying the design process, reducing errors, and standardizing constraint usage across tools. Such a solution would relieve engineers from the repetitive and detail-oriented aspects of constraint definition, freeing their time and cognitive effort to focus on higher-level innovation and strategic design decisions.

1.3 Research Objectives

The central objective of this dissertation is to design and implement an intelligent system that automates the generation of design constraints within ASIC digital design workflows, leveraging the latest advancements in generative AI. This research was conducted in collaboration with Synopsys, a leading Electronic Design Automation (EDA) company specializing in silicon design and verification, silicon Intellectual Property (IP), and quality solutions. By embedding the proposed approach within the company's workflow, the project supports the industry shift toward intelligent automation in digital design [3]. To achieve this primary objective, the dissertation is structured around several specific objectives:

1. Prepare a Set of Designs

- (a) Create a series of digital designs, using Synopsys tools.
- (b) Use this dataset to test and refine constraint generation, offering a controlled environment for evaluating initial system outputs.

2. Develop a Retrieval-Augmented Generation (RAG) Based system with Prompt Engineering Techniques

- (a) Build a system that leverages pre-trained LLMs enhanced by RAG and customized prompt strategies.
- (b) Refine these prompts to improve the precision and reliability of the constraints generated by the RAG-based prototype.

3. Implement a comprehensive evaluation methodology

- (a) Develop a methodology to rigorously measure the performance and reliability of the proposed approach.
- (b) Apply this methodology to a range of benchmark designs, including those with multiple clock domains and complex timing requirements, in order to assess not only the technical accuracy of the generated constraints but also their practical applicability in real verification and synthesis workflows.

1.4 Research Contributions

The dissertation contributes to the integration of generative AI into the ASIC digital flow. The primary contribution of this dissertation is to design and implement a RAG-based prototype to address constraint generation in ASIC workflows by combining a curated knowledge base of tool documentation with prompt-engineered LLMs. Rather than proposing a fixed system, this work investigates combinations of different models, prompting strategies, and retrieval configurations.

Moreover, to ensure that the generated constraints can be meaningfully assessed, the dissertation will define a comprehensive evaluation framework that seeks to evaluate not only syntactic correctness, but also the practical usability in verification and synthesis contexts of these.

In doing so, the dissertation will lay the conceptual foundations for integrating AI-driven constraint generation into real ASIC workflows. While full industrial integration may fall outside the immediate scope of this research, the work will outline a pathway for future systems to incorporate direct feedback from tools, enabling the creation of AI-assisted, self-improving design environments that progressively reduce the manual effort required from engineers.

1.5 Research Questions

The research presented in this dissertation is primarily based on two fundamental questions which guide its trajectory and shape its practical relevance. These questions not only reflect the technical challenges inherent in the project but also emerge from the broader implications for real-world design environments.

- **RQ1: How effective are LLMs, when combined with RAG and prompt engineering, in generating accurate and usable design constraints?** This question explores the technical viability of using generative AI to automate ASIC digital design workflow.
- **RQ2: How can such a tool be seamlessly integrated into existing ASIC design workflows to ensure it is both practical and valuable for engineers?** This second question focuses on usability and adoption. The goal for this question is to move beyond a theoretical solution and instead propose an integration strategy that respects the realities of engineering work.

1.6 Dissertation Structure

The document is organized into six more chapters. Chapter 2 introduces the background knowledge of ASIC digital design. Chapter 3 introduces the fundamental concepts of LLMs and explores their emerging role in the EDA domain. Chapter 4 presents the background of RAG, outlining its workflow, benefits, and limitations. Chapter 5 outlines the implementation details of the proposed solution, including system architecture, benchmark design development, and the rationale behind framework, database, and model selection. Chapter 6 presents the results and analysis of the tested cases, combining quantitative metrics, inference time, tool-based inspection, and expert evaluation. Finally, Chapter 7 concludes the dissertation by summarizing the main contributions and proposing potential directions for future research.

Chapter 2

ASIC digital design

This chapter introduces the role of design constraints within the ASIC digital design flow, highlighting how they guide tools and processes from Register Transfer Language (RTL) to final verification. It explores how constraints ensure correct functionality, timing closure, and reliable operation across different stages of the workflow. Special attention is given to challenges in multiple-clock designs, including Clock-Domain Crossing (CDC) and Reset-Domain Crossing (RDC) issues, setup and hold-time constraints, and the phenomenon of metastability, critical factors in ensuring robust and functional silicon.

2.1 Constraints in the ASIC design workflow

The journey from a high-level system specification to a fabricated ASIC involves a series of carefully coordinated steps. Throughout this process, design constraints play an essential role, being more than just technical parameters; they act as rules, guiding tools and decisions at every phase to ensure the chip behaves as intended under real-world conditions. The key steps at which design constraints are applied are the following [4]:

1. **RTL Design and Verification:** In the early design phase, constraints define functional expectations and help shape the digital behaviour of the chip. Typical constraints include clock frequencies, reset logic, and basic Input/Output (I/O) timing. These guide simulation and help verification tools evaluate whether the RTL meets its specification.
2. **Synthesis:** During synthesis, constraints become more specific, dictating how the RTL should be transformed into a gate-level netlist. Timing constraints, such as setup and hold times, are crucial here. They ensure that the synthesized design can operate correctly at the desired clock frequency.
3. **Physical Design and Implementation:** As the focus shifts toward layout, constraints are fine-tuned to address the realities of chip geometry. Placement constraints define where components can or cannot be located, while clock tree synthesis rules ensure balanced clock

distribution. Routing constraints also help maintain signal integrity, prevent crosstalk, and enforce design rules that are crucial for manufacture.

4. **Timing Analysis and Final Verification:** In the final stages, timing constraints are indispensable for verifying that the chip will operate reliably across all corners of process, voltage, and temperature (PVT). Static Timing Analysis (STA) tools use these constraints to check setup and hold times, detect violations, and confirm that timing closure is achieved [5].

To better illustrate how constraints interact with the digital design process in practice, consider the simplified constraint-driven flow shown in Figure 2.1.

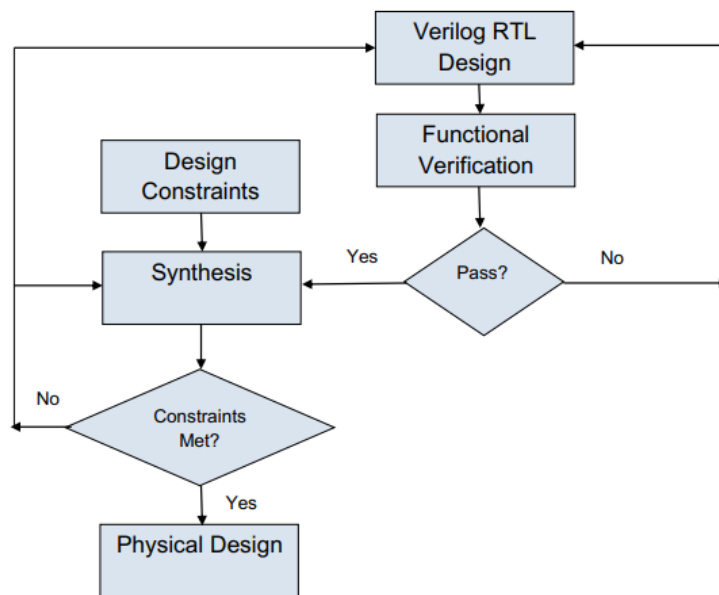


Figure 2.1: Constraint-driven flow in ASIC design (from [4]).

2.2 Multiple-clock designs

Designs that operate with a single clock are straightforward and simple to implement. However, in practical scenarios, very few designs rely exclusively on a single clock [4]. Instead, most systems involve multiple clock domains, which introduces additional complexity. This section explores the challenges associated with multiple-clock designs and introduces solutions not only at the design level but also through the use of design constraints, ensuring a robust and reliable implementation.

In a single-clock domain design, a single clock signal drives the entire circuit; by contrast, in a multiple-clock design, as illustrated in Figure 2.2, different parts of the system are governed by independent clocks. This multi-clock approach introduces challenges related to data integrity and synchronization whenever signals must traverse shared boundaries. CDCs arises in such designs when a signal generated in one clock domain is sampled in another, while RDCs refers to similar

synchronization issues that occur when signals transition between domains controlled by different reset signals. These crossings inherently carry the risk of setup and hold time violations, caused by the asynchronous relationship between the clocks or resets. Effectively addressing these risks requires careful application of synchronization techniques, along with thorough CDC and RDC verification. Independent studies confirm that Synopsys' CDC verification solution, VC SpyGlass, plays a crucial role in mitigating these issues, significantly reducing the likelihood of CDC-related silicon failures and thereby improving design reliability, performance, and minimizing costly re-spins [6].

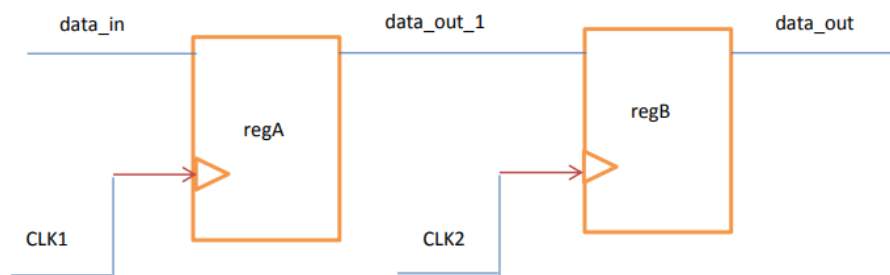


Figure 2.2: Multiple clock system (from [4]).

2.2.1 Setup and hold-time constraints

Setup and hold violations are the foremost timing issues in digital designs since they have a direct effect on the reliable operation of the flip-flops (FF) in the circuit. These happen when the timing requirements at the data input of the FF are not satisfied concerning the clock signal.

The setup time is the minimum time before the active edge of the clock during which the data input (D) must remain stable. In case the data is altered just before the clock edge, the FF can be unable to retain the actual value, which in turn will result in a setup violation. Conversely, the hold time is the least time beyond the active edge of the clock when the data input (D) has to stay unchanged. If the data changes too soon after the clock edge, the FF might not properly register the value, causing a hold violation. Both time constraints are represented in Figure 2.3.

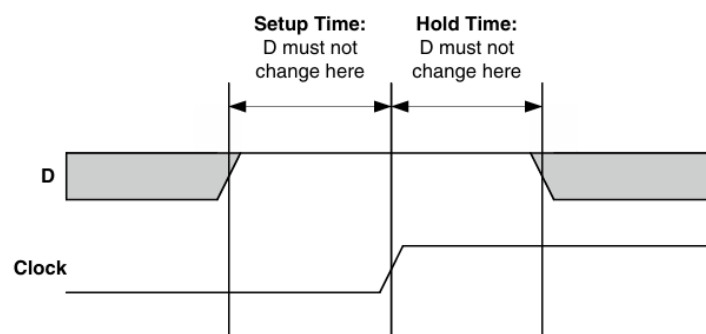


Figure 2.3: Setup and hold-time (from [7]).

The diagram in Figure 2.4 illustrates two clock domains, *xclk* and *yclk*, along with two CDC signals from *xclk* domain, *xclk_output1* and *xclk_output2*. A timing violation occurs when the signal *xclk_output1* is sampled within the *yclk* clock domain. If *xclk_output1* flips close to the rising edge of *yclk*, this can cause both violations regarding the *yclk* signal. Consequently, the signals in the *yclk* domain that depend on *xclk_output1* can reach a metastable condition, which may lead to erroneous operations. In contrast, the signal *xclk_output2* is sampled within the setup and hold timing requirements of clock *yclk*, thereby entirely avoiding metastability issues.

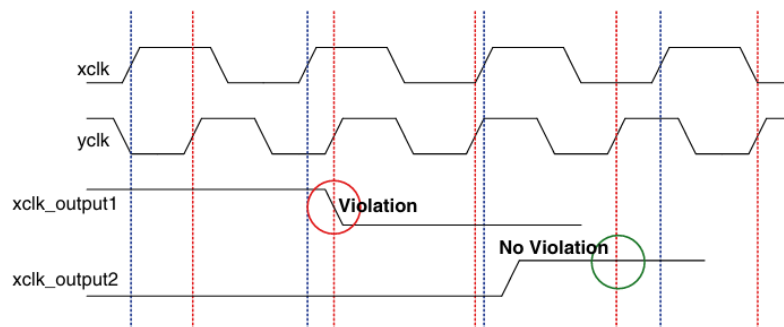


Figure 2.4: Timing diagram for a dual clock system (from [7]).

2.2.2 Metastability

Metastability occurs when one of the time requirements of a FF is violated. The combination of both setup and hold time determines the width of the metastability window. If the input signal changes within this time window, the FF may enter an undefined or metastable state. This metastable condition can result in glitches or a temporarily unstable output, which can take an extended time to stabilize and reach a defined state. The fundamental values employed to illustrate a FF's metastability properties include:

- T_0 : This is the likelihood that the device will enter a metastable state.
- τ : This is the metastability time constant which is indicating the exponential decay of the metastable state respectively.
- h : The average propagation delay for the specific device.

A detailed study conducted by Philips Semiconductors [8] illustrated how these parameters can be extracted through controlled experimentation. In their work, a high-frequency asynchronous data signal was used to induce metastable states in a FF, and the number of failures, cases where metastability persisted beyond a certain time threshold, was measured. These failures followed an exponential decay, allowing τ to be calculated from the slope of the failure curve on a semi-logarithmic plot. T_0 was derived by extrapolating this data back to time zero, effectively estimating the size of the metastability window at the clock edge.

Due to the exponential nature of metastability resolution, designers can mitigate the risk by incorporating a safety margin after the standard propagation delay before reading the FF's output.

This additional wait time significantly reduces the probability of encountering a metastable failure. There is still an infinitely small chance, even after taking the necessary steps, that the FF will continue to stay in a metastable condition forever.

The inevitability of metastable behaviour in bistable systems has also been rigorously formalized in Leonard R. Marino, which extends the theoretical framework to encompass both synchronous and asynchronous systems. Marino's analysis confirms that metastable conditions are not merely anomalies but intrinsic characteristics of digital systems interfacing with asynchronous inputs [9].

Chapter 3

Large Language Models

This chapter equips the reader with the conceptual and practical foundations of large language models that underpin the proposed system. Since the approach delegates format-sensitive constraint generation to prompted LLMs, it is necessary to clarify how transformer models represent text, how prompting strategies shape reasoning, and how different model families compare on relevant benchmarks. These foundations justify the prompt design presented in Chapter 5 and explain inherent limitations such as context-window size that influence chunking and evaluation. The chapter also surveys prior applications of LLMs to EDA, highlighting both strengths and common failure modes.

3.1 Terms and Definitions

This dissertation explores technical concepts associated primarily with natural language processing (NLP) and machine learning (ML). Thus, it is essential to first set the definitions of these fundamental terms. NLP is a field of study in artificial intelligence focused on the relation between computers and humans. In this area, language models (LMs) aim to provide computers with the abilities to process, understand and generate human languages.

A LLM consists in a probabilistic model that is trained to predict the likelihood of word sequences. These models are able to perform tasks such as text generation, translation and summarization by capturing contextual information across sentences. As a result of the development in this area, language models have undergone a tremendous growth in their size and complexity leading to large neural-based systems with billions of parameters.

These models are not only characterized by the enormous databases used for their training but also by their massive internal structures, which can contain hundreds of billions of parameters. Parameters are the internal variables learned from the training data that define the model’s behavior and capability: model GPT-4, with more than 175 billion parameters, exemplifies the scale of modern language models [10].

The basic units of text processed by language models are called tokens. A token is a piece of text that can be a word, sub-word, character, or punctuation mark, depending on the tokenization

scheme being used. The application of this abstraction enables language models to process various languages and manage large vocabularies efficiently.

In order to process each token, language models use embeddings, which are numerical vector representations that encompass the semantic and syntactic qualities of linguistic elements. For example, the word "tokenization" could be tokenized into "token" and "ization" as subwords. These embeddings facilitate the model to carry out operations in a continuous vector space where word relationships are quantifiable [11].

Embeddings are dense vector representations of tokens designed to capture semantic meaning by position words with similar meanings close to each other in a high-dimensional space. They allow language models to understand relationships between words, enhancing their ability to generate coherent and contextually relevant text [12].

3.2 Evolution of LLMs

The evolution of LLMs has been marked by significant advancements that improved their capability to understand and generate language.

Initially, Statistical Language Models (SLMs) were the entry point for the computational language comprehension. These predictive models, including n-gram models, estimated the probability of a certain word based on the frequency of the previous words in that sentence. However, SLMs had limitations such as data sparsity and incapability to capture long-range dependencies.

With the development of neural networks, more sophisticated language models emerged, such as Neural Language Models (NLMs). Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, which allowed language models to process sequences with the awareness of the preceding context, overcoming the limitations of the previous models [13]. NLMs introduced embeddings, which allowed models to understand semantic relationship between words [14]. Nevertheless, they faced challenges such as the vanishing gradient problem, which affected their learning of dependencies over longer sequences.

The arrival of the Transformer architecture made a significant change in language models [12]. Transformers, by virtue of self-attention mechanisms, allowed the models to assess the significance of the various words present in a sequence at the same time, instead of in a sequential manner. The parallel structure improved computational efficiency and the model's ability to handle long-range dependencies, leading to the emergence of Pre-trained Language Models (PLMs) such as BERT [15] and GPT [16] which could be trained on a large corpora and fine-tuned on specific tasks for greater outcomes [17]. The introduction of LLMs following PLMs was the result of their enormous size in parameters and training data.

LLMs are advanced LMs with a massive scale, often containing billions of parameters and trained on extensive amounts of text data. Building on the Transformer, the latest generation of LLMs exhibit emergent capabilities, as is described in section 3.5.

3.3 Transformer Architecture and self-attention mechanism

The Transformer architecture represents a shift in the field of NLP, redefining how computers interpret and generate human language. While former models such as RNNs and LSTMs worked sequentially, the Transformer operates on entire sequences in parallel. This approach offered significant improvements in both computational efficiency and linguistic understanding.

At the core of the Transformer architecture is the self-attention mechanism. This process enables the model to weigh the importance of each token relative to others in a sequence. Instead of relying on fixed windows or sequential memory, the model assigns dynamic weights to tokens based on their contextual significance. Each token is transformed into query, key, and value vectors. The query vector encodes what information each token seeks from the rest of the sequence, the key vector indicates what kind of information a token can provide and the value vector carries the actual information content of a token. These vectors are then used to compute attention scores, determining how much focus each token should allocate to the others.

To enrich this capability, the Transformer employs multi-head attention. This involves multiple attention heads operating in parallel, each with its own set of query, key, and value matrices. Each head learns to focus on different relationships within the data, some may attend to syntactic structures, while others capture semantic nuances. The outputs from these heads are concatenated and linearly transformed, giving the model a more nuanced understanding of token dependencies.

However, because the Transformer processes tokens simultaneously, it lacks a built-in sense of order. To compensate, positional encodings are appended to the token embeddings, carrying information about token positions. Positional encoding typically utilizes a sinusoidal function, enabling the model to differentiate the tokens based on their position.

Each Transformer layer applies a feedforward network independently to the output of the multi-head attention mechanism. Residual connections are also incorporated to prevent the vanishing gradient problem, where error gradients shrink as they propagate back through the network's layers, which can halt learning, and to ensure stable gradient flow during training.

The main stream of architectures of LLMs can be categorized in encoder-only, decoder-only and encoder-decoder models.

Encoder-only LLMs, like BERT, only use bidirectional training, allowing them to understand the context of a token by integrating information from both sides of the input sequence, making them ideal for tasks such as document classification.

Decoder-only LLMs, including GPT, process text unidirectionally (left-to-right). They use a single text stream as input and conditioning is simply based on past tokens.

Encoder-decoder LLMs, combine both approaches, using an encoder to understand the input sequence and a decoder to generate the output, making them suitable for tasks requiring comprehension and generation of text.

LLMs are adaptations of the Transformer architecture with increased layers, parameters and training corpora. According to "Scaling Laws" model performance tends to improve as any of

these elements increases: model size, depth, or data volume. While OpenAI's scaling laws suggest priority in model size over data volume [18], Google DeepMind's recommend increasing model and data sizes proportionally [19]. These scaling laws inform how researchers allocate computational resources and predict performance improvements.

3.4 Fine-tuning and Prompting techniques

3.4.1 Fine-tuning Strategies

Fine-tuning is the subsequent stage at which a pre-trained model is further trained on a smaller, task-specific dataset. The goal of fine-tuning is to adapt the generalized knowledge acquired during pre-training to perform optimally on a particular target task. This process can be achieved by adjusting the parameters of the pre-trained model.

Full fine-tuning consists in updating all parameters of the pre-trained model. This technique is effective, but computationally demanding for very large models. Parameter-Efficient Fine-tuning (PEFT) methods, such as Low-Rank Adaptation (LoRA), Quantized Low-Rank Adaptation (QLoRA), and adapters, selectively update a small subset of the model's parameters or introduce new trainable modules. This approach minimizes both the computational costs and the memory usage, making it feasible to personalize very large LMs on weaker hardware.

1. **LoRA:** Rather than modifying the full weight matrices of the transformer layers, LoRA represents each weight update as the product of two much smaller matrices. The base model's parameters remain frozen during fine-tuning, and only these low-rank matrices are learned.
2. **QLoRa:** Building on LoRA's low-rank factorization, QLoRA first quantizes the frozen pretrained model to 4-bit precision, drastically shrinking its memory footprint. The low-rank adapters are then applied to this quantized backbone. Combining extreme precision reduction with LoRA's parameter-efficient updates, QLoRA enables full-scale fine-tuning.
3. **Adapters:** Adapters introduce small, trainable bottleneck layers between the existing sub-layers of the Transformer. Only these adapter parameters are updated during fine-tuning, while the core model weights are kept fixed.

Collectively, PEFT methods achieve substantial reductions in training time and resource requirements, all while preserving the accuracy of tasks.

Instruction tuning is a technique where a pre-trained LLM is further trained on a dataset consisting of high-quality (instruction, output) pairs. The "instruction" is a natural language prompt that specifies a task, and the "output" is the desired response that follows that instruction.

Reinforcement Learning from Human Feedback (RLHF) is a technique that aligns LLMs with human preferences that are difficult to capture with simple instruction-output pairs. Used in models such as Instruct-GPT and ChatGPT, it leverages human judgments as a reward signal to refine the LLM's behaviour.

3.4.2 Prompt Engineering

Prompt engineering is an essential method in optimizing LLMs to generate accurate, task-specific responses. This technique involves "programming in natural language" by carefully designing input prompts to guide the model in producing desired, accurate and relevant outputs.

While fine-tuning is less resource-intensive than pre-training, it still requires model training and the availability of task-specific datasets, leading to considerable computational costs and manual effort. Contrarily, prompting methods efficiently adapt general LLMs to specific domains. Common prompting techniques include In-Context Learning (ICL) [20], Chain-of-Thought (CoT) [21], and RAG [22], allowing for effective alignment of the model's capabilities with user requirements without the need for extensive retraining.

ICL refers to the ability of sufficiently large pre-trained language models to learn new tasks, or adapt to specific formats, by simply being provided with examples within the input prompt itself, without any explicit parameter updates or fine-tuning of the model's weights. It's often referred as "few-shot learning" when a small number of examples are provided.

This technique is an emergent ability of LLMs, meaning it becomes significantly more pronounced and effective as model size and pre-training data scale up. Smaller models might struggle with ICL, while larger models demonstrate remarkable proficiency [20]. ICL can be categorized into three primary types:

1. **Zero-Shot ICL:** No examples are provided.
2. **One-Shot ICL:** One example is provided to illustrate the task.
3. **Few-Shot ICL:** A small number (2-5) of examples are provided to demonstrate the task.

This allows LLMs to perform diverse tasks without requiring task-specific fine-tuning, significantly reducing development time and computational resources. It remains effective even with very few labelled examples, and new tasks can be handled quickly by simply crafting a new prompt. Moreover, since the model parameters aren't updated, the model's broad knowledge from pre-training is maintained.

ICL also shares a standard limitation with other prompt-based techniques: the model's fixed context window. This strict upper limit on the total number of input tokens restricts the number of examples, instructions, and any auxiliary material within the prompt. Moreover, the model's performance can be susceptible to the order, format, and wording of the examples and instructions. Although powerful, ICL may not achieve the same level of performance as fine-tuning on a large dataset and it lacks a comprehensive theoretical understanding of its underlying mechanisms.

CoT is a specific prompt engineering technique that encourages LLMs to generate a series of intermediate reasoning steps before arriving at a final answer. Instead of expecting a direct solution, the prompt guides the model to "think step-by-step", mimicking a human's thought process when solving complex problems [21]. There are three primary forms of CoT prompting:

1. **Zero-Shot CoT:** Simply adding such as like "Let's think step-by-step" to prompts can elicit CoT reasoning from sufficiently large models, without the need for explicit examples.

2. **Few-Shot CoT:** The original formulation provides a few illustrative examples consisting of questions, detailed reasoning steps, and answers.
3. **Auto-CoT:** An automated method that selects diverse examples from a dataset, generating their reasoning chains to construct a few-shot CoT prompt [23].

The advantages of CoT prompting include increased accuracy in complex tasks, boosting the performance in reasoning intensity tasks. Additionally, it can enhance interpretation by providing insight into how the model arrived at its answer, and it forces the model to generate intermediate steps. However, a downside of CoT prompting is the increase of token count due to the generations of those intermediate steps, which consequently extends inference time. Furthermore, this technique is most effective when utilized with larger LLMs. Finally, RAG consists in integrating retrieved context from an external knowledge database source in the LLM prompt. This technique is detailed in the next Chapter.

3.5 Trends in Leading Large Language Models

The landscape of LLMs is rapidly evolving, with the focus shifting from simple scale increases to innovations in reasoning, efficiency, and accessibility. This new approach is shaping the future of artificial intelligence.

3.5.1 Advanced Reasoning and Thinking Systems

The latest models are moving beyond simple text generation, incorporating complex systems that allow them to solve complex problems and are suitable for advanced AI agents. Anthropic's Claude Opus 4 is a prominent example, with a hybrid reasoning system that enables two distinct operational modes: near-instant responses and an "extended thinking mode" for complex tasks. This model achieved impressive results on programming benchmarks, including 72.5% on SWE-bench and 43.2% on Terminal-bench, and can maintain consistent performance on long-duration tasks [24].

Similarly, Mistral AI introduced Magistral Medium and the open-source Magistral Small [25]. Designed for transparent, multi-step logic, the 24B-parameter Magistral Small builds on Mistral Small 3.1 with enhanced reasoning capabilities. Despite its compact size, Microsoft's Phi-4-reasoning-plus also stands out in this field. This 14B-parameter open-weight model, fine-tuned from Phi-4 using meticulously curated CoT datasets and RL, surpasses larger models, such as OpenAI's o1-mini and DeepSeek-R1-Distill-Llama-70B across various benchmarks, including mathematics and PhD-level science questions [26].

3.5.2 Optimization and Cost Efficiency

The evolution of LLMs is increasingly driven by the need to reduce costs and improve efficiency [27]. OpenAI responded to this challenge with GPT-4o, a family of models that achieved a remarkable 83% reduction in operational costs and substantial latency improvements compared to its predecessor, GPT-4o, while matching or exceeding its performance [28]. Another key innovation is the Mixture-of-Experts (MoE) architecture, central to Alibaba Cloud's Qwen3 models. The flagship Qwen3-235B-A22B model, with a total of 235 billion parameters, activates only 22 billion per generation step, optimizing computational efficiency. It also features the ability to switch seamlessly between a "thinking mode" and a "non-thinking mode," and its smaller MoE model, Qwen3-30B-A3B, outperforms QwQ-32B, a larger dense model with ten times fewer activated parameters [29].

3.5.3 Lightweight and Portable Open-Source Models

Meta and other companies have democratized access to high-quality models through open-source releases. The Gemma 3 family [30], developed by Google DeepMind, is a prime example. Built on the same technological foundations as the Gemini models, these open-weight models are designed for speed, portability, and responsible AI development. The Gemma 3 family includes models of various sizes, with a distinctive feature being Quantization-Aware Training (QAT), which allows models like the Gemma 3 27B to run locally on consumer GPUs such as the NVIDIA RTX 3090 while maintaining high quality. Meta's Llama 3.1 [31] is a multilingual open-source model that incorporates Supervised Fine-Tuning (SFT) and RLHF to align better with human preferences. Its successor, Llama 3.3 70B, was developed to match the performance of a much larger model with a significantly smaller computational footprint. Mistral AI's Magistral Small, with its 24B parameters and 128K-token context window, was explicitly designed for local deployment. It can run on a single RTX 4090 GPU or a 32GB RAM MacBook when quantized [32], highlighting the growing importance of portability and accessibility for deploying AI in various environments.

3.6 Performance Benchmarks

The following benchmarks provide a standardized quantitative measure of language models in terms of intelligence and capability across diverse domains. These evaluations offer insight into how well models perform in real-world tasks, academic challenges, and complex reasoning scenarios.

One of the most widely adopted benchmarks, the Massive Multitask Language Understanding (MMLU) score [33] evaluates the performance of LLMs across a wide range of subjects and tasks; it measures how well a model can answer multiple-choice questions, typically in zero-shot or few-shot learning scenarios, across 57 different subjects. MMLU assesses a model's ability to understand and reason about various topics, serving as a strong indicator of general academic knowledge and cognitive flexibility.

Complementing this general academic focus, the Software Engineering Benchmark (SWE-bench) score [34] is a benchmark used to evaluate LLMs performance on real-world software engineering tasks. It tests how well these models can generate code that resolves issues in popular open-source Python repositories. A higher SWE-bench score indicates that the LLM is more capable of understanding, modifying, and testing code to fix problems, as demonstrated by passing the same unit tests as the original pull request.

Similarly, the Terminal-bench score [35] focuses on evaluating an agent's ability to perform complex tasks in a terminal environment. This benchmark offers both a framework and a dataset for quantifying how well a LLM or agent interacts with command-line interfaces, reflecting its practical proficiency in system-level operations.

Shifting the focus toward mathematical reasoning, the AIME Score [36] is derived from the The American Invitational Mathematics Examination, a prestigious mathematics competition for top-performing high-school students. The benchmark involves 15 questions of increasing difficulty, with the answer to every question being a single integer from 0 to 999. It evaluates a model's advanced mathematical reasoning skills and symbolic problem-solving abilities.

Finally, the HumanEval benchmark [37] is widely used to assess the functional correctness of code generated by LLMs. It focuses on evaluating whether the code solutions provided by the model produce correct outputs when executed against unit tests, making it a valuable tool for measuring performance in code generation tasks.

3.7 Large Language Models in Electronic Design Automation

As detailed in the previous sections of this chapter, the Transformer architecture, the mechanisms of pre-training and fine-tuning have enabled LLMs to acquire vast knowledge and skills extending into the technical domain. This potential has been increasingly explored in EDA, promising to transform hardware design methodologies.

3.7.1 Current State of LLM Integration in EDA

The emerging applications of LLMs in EDA encompass a diverse range of tasks that traditionally require intensive human intervention, from Hardware Description Language (HDL) generation to debugging and optimization. Several notable frameworks have emerged, each addressing different aspects of the EDA workflow while revealing both the potential and limitations of LLM integration in hardware design.

Chip-Chat by Blocklove et al. [38] represents one of the early investigations into LLM capabilities for EDA, specifically examining how models such as ChatGPT and Gemini perform in generating Verilog code from textual descriptions. This work demonstrates both the potential and challenges in obtaining functional HDL from natural language specifications, highlighting fundamental accuracy concerns where LLMs can generate plausible but incorrect code, a phenomenon known as "hallucination" that is particularly problematic in hardware design contexts.

Building upon these initial findings, *ChipGPT* [39] proposes a more sophisticated four-stage framework that addresses many of the limitations identified in earlier work. The framework employs a structured approach with manual specification division: a "Prompt Manager" for optimized information presentation, an "Output Manager" for error correction through both machine and human feedback, and a "Cost Manager" for Power, Performance and Area (PPA) optimization. This work explicitly acknowledges the challenges of direct LLM integration due to vague inputs and outputs, and the lack of refined prompt templates for high-quality HDL generation, while proposing solutions through careful pre and post-processing control.

ChatCPU [40] focuses specifically on CPU design and verification, implementing an end-to-end approach that guides the process from initial specifications to final RTL implementations. The platform incorporates LLM fine-tuning and processor description languages for CPU design automation, representing a paradigm where the LLM becomes more deeply integrated and specialized through domain-specific training. However, the work acknowledges that existing LLMs exhibit significant challenges in HDL design and verification, particularly for complicated hardware systems.

Addressing the critical issue of debugging, *RTLFixer* [41] presents a framework that combines LLMs with RAG and ReAct agent to automatically correct syntax errors in Verilog code. ReAct agent is an AI agent that uses "reasoning and acting" framework to combine CoT reasoning with external tool usage. This hybrid approach demonstrates the effectiveness of LLMs as autonomous agents for interactive debugging, where RAG provides relevant context and ReAct enables iterative problem-solving through action and observation cycles. The framework exemplifies how LLMs can excel at solving complex problems through iterations while reducing direct human intervention.

VeriPPA [42] tackles the fundamental challenge that LLMs lack inherent knowledge about critical PPA metrics in chip design. The framework implements a two-phase refinement protocol specifically aligned with Power, Performance, and Area benchmarks: the first stage improves operational and linguistic accuracy, while the second stage aligns code with PPA requirements through feedback loops with analysis tools. This work illustrates how external feedback mechanisms are essential for guiding LLMs toward optimal hardware design solutions.

3.7.2 Summary of Approaches and Future Directions

The examination of these frameworks reveals several distinct approaches for LLM integration in EDA. Controlled integration approaches, exemplified by *ChipGPT* and *VeriPPA*, impose significant pre and post-processing control to mitigate LLM output ambiguity and integrate engineering metrics that LLMs don't inherently understand. While these approaches offer higher reliability and optimization capabilities, they require more manual effort and sophisticated engineering frameworks.

Autonomous agent approaches as demonstrated by *RTLFixer*, leverage RAG and iterative feedback loops to enable LLMs to reason and act independently in debugging scenarios. This

paradigm excels at complex problem-solving through iterations, but faces scalability challenges for very complex bugs and systems.

Specialization through fine-tuning, as explored in ChatCPU, offers the potential for improved accuracy and domain understanding, but comes with computational costs and requirements for large, domain-specific datasets. This approach addresses the limitation that LLMs, despite being pre-trained on vast corpora, may lack sufficient exposure to highly specific hardware design data.

The summary of these approaches suggests that effective LLM integration in EDA requires hybrid strategies that combine explicit control mechanisms, retrieval-augmented context provision, and iterative feedback loops. Such integrated approaches could represent a promising path toward automating complex EDA tasks, including design constraint generation, effectively and reliably. However, fundamental challenges remain, including computational intensity, data privacy concerns with cloud-based LLMs, and the need for deep understanding of computer architecture and digital electronics that current LLMs struggle to fully capture.

Chapter 4

Retrieval-Augmented Generation

This chapter proposes Retrieval-Augmented Generation as the core mechanism for grounding model outputs in tool manuals and design-specific context. As the objective is to leverage pre-trained LLMs enhanced by RAG, the pipeline is formalized in terms of indexing, querying, retrieval, and generation, together with its benefits and operational trade-offs. The chapter also discusses limitations of RAG, which informed the implementation choices adopted in this work and the evaluation methodology, present in Chapter 5. Including this chapter is therefore essential to interpret the ablation studies reported in Chapter 6 and to understand why the observed improvements, reduced "hallucination" and stronger alignment with tool-specific constraints, arise specifically from RAG.

4.1 Introduction to RAG

Retrieval-Augmented Generation is a hybrid approach that improves the performance of LLMs through the integration of the information dynamically retrieved from the external knowledge source. In essence, RAG consists of a retrieval mechanism and a generative model in a pipeline that can be divided into four broad categories [43]:

1. **Indexing:** Document preprocessing involves breaking the documents into smaller textual units, followed by utilizing models such as Sentence-BERT [44] for the creation of dense vector embeddings. These embeddings capture semantic meaning and are stored in a vector database.
2. **Querying:** When a user submits a query, it is encoded into an embedding using the same model employed during indexing. This ensures compatibility between the query vector and document embeddings for meaningful comparison in the semantic space.
3. **Retrieval:** The embedded query is compared against the stored document vectors using similarity metrics such as cosine similarity or Euclidean distance. The system ranks and retrieves the top-k most semantically aligned segments. Some advanced RAG systems employ

multi-step retrieval to iteratively refine search results, or re-ranking techniques to ensure the most relevant content is prioritized [45].

4. **Generation:** The retrieved content is concatenated with the original query and passed to a generative model, typically a transformer-based LLM. Attention mechanisms within the model enable it to focus on and incorporate pertinent elements of the retrieved text, generating outputs that are both coherent and contextually accurate. Enhancements such as cross-attention and modular co-optimization further improve the fidelity and fluency of the generated response.

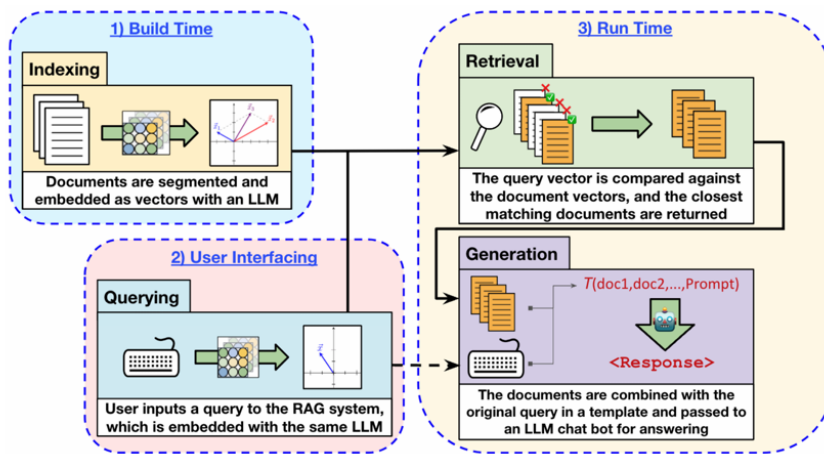


Figure 4.1: RAG flow (from [43]).

This process is illustrated in Figure 4.1 and it outlines how RAG combines these four steps to create more informative and context-aware outputs.

The transformation of RAG systems led to multiple variations, such as *Naive RAG*, which uses the simplest steps of retrieval and generation, *Advanced RAG*, which adds reranking and more selective filtering mechanisms, and *Modular RAG*, where the retrieval and generation modules are more integrated and jointly optimized for better performance in several tasks [46].

4.2 Benefits and Applications of RAG

RAG provides a great solution to help with the enhancement of the LLMs, particularly in issues that demand up-to-date, trustworthy, and specific knowledge of the context. Besides, through its retrieval of the external real-time information RAG deals precisely with the lack of ability of static LLMs to get access or combine data outside what they have been trained on.

A principal advantage of RAG is its capacity to improve factual accuracy. Generative models augmented with external retrieval are less prone to the phenomenon of "hallucination", as they can ground their responses in verifiable sources [47]. This has proven especially beneficial in open-domain question answering (OpenQA), where factual integrity is essential.

In addition to being accurate, RAG also adds to the specificity and relevance of the produced content. It provides the opportunity for models to use domain-specific repositories, which is its main advantage in areas such as healthcare [48] and finance [49] where fresh and specialized information is imperative. Unlike traditional LLMs that depend solely on general corpora, RAG systems can dynamically adapt to the demands of rare or emerging topics without requiring extensive retraining.

RAG’s architectural flexibility enables easy integration across a wide range of real-world use cases. For example, in customer support, RAG-powered chat-bots can fetch the latest responses from dynamic query logs to ensure more accurate and contextual replies [50]. Another application of RAG in action is Google’s NotebookLM.

NotebookLM is an AI-powered research tool and “virtual research assistant” developed by Google Labs [51]. The tool has demonstrated tangible advantages in specialized domains, for example, it achieved 86% diagnostic accuracy in lung cancer staging tasks when provided with proper reference documentation, significantly outperforming GPT-4 on the same task, and retrieved accurate citations to support its outputs [52].

Moreover, RAG reduces the need for repeated fine-tuning, offering a more computationally efficient alternative for scenarios requiring frequent updates. This efficiency makes it both scalable and cost-effective for enterprises managing rapidly changing knowledge environments.

Overall, the combination of real-time retrieval and generative modeling empowers RAG-based systems to produce more accurate, contextually appropriate, and trustworthy outputs.

4.3 Optimization of RAG

To develop a high-performing RAG system, various optimization strategies must be applied across the entire pipeline [43]. The indexing phase begins with text preprocessing, transforming documents into clean, plain-text formats while removing extraneous or irrelevant content to ensure compatibility with LLM-friendly inputs. Effective text chunking is crucial: documents should be divided into segments that strike a balance between maintaining semantic coherence and adhering to model input constraints. Metadata such as titles, timestamps, and authorship can also be appended to enhance retrieval precision and contextual richness during embedding. Equally important is the selection of an embedding model that offers a robust trade-off between computational efficiency and semantic depth. Models evaluated in benchmarks such as the Massive Text Embedding Benchmark (MTEB) [53] provide valuable guidance in choosing the right tool, particularly considering their impact on context window limits and chunk sizing.

In the retrieval stage, one must select a vector database which aligns with the system’s needs for speed, scalability, and resource efficiency. Options are described in section 5.3. Tuning retrieval parameters, particularly the number of top results, is essential for optimizing the contextual payload passed to the LLM: too few results may miss critical data, while too many can overwhelm the model’s input buffer. The choice of similarity metric, such as cosine similarity or Euclidean

distance, should be compatible with the structure of the embedding space and optimized for relevance scoring.

Finally, in the generation phase, selecting an appropriate LLM involves balancing response quality with computational cost. Prompt design also plays a significant role in performance: prompts should be constructed to integrate the query and retrieved content in a logically coherent and interpretable manner, thereby enabling the model to leverage contextual information effectively.

Together, these optimizations ensure that RAG systems are not only accurate and context-aware but also enable efficient and scalable deployment.

4.4 Current challenges and Limitations of RAG

While RAG has emerged as a promising paradigm for enhancing the factuality and adaptability of language models, its actual application is far from trivial. With further examination, it is possible to find some persistent challenges regarding infrastructure, algorithmic design, and ethical considerations, among others.

One of the most pressing concerns is scalability. RAG systems rely on the retrieval of external knowledge from large, often evolving corpora. As these datasets grow, ensuring fast, memory-efficient retrieval becomes increasingly difficult. The computational cost of maintaining, querying, and updating massive vector stores can be prohibitive, especially in real-time applications or environments with constrained hardware.

Equally critical is the issue of retrieval quality. A RAG model can only be as good as the context it retrieves, and current retrieval systems may struggle to surface the most relevant information. Retrieval may yield documents that are outdated, tangential, or outright irrelevant, introducing noise into the generation pipeline. Improving retrieval precision remains an active area of research [54] [55].

Another subtle, yet impactful limitation, is the lack of seamless integration between the retrieved knowledge and the generated output. Despite having access to relevant information, RAG models may struggle to weave this content into a coherent and contextually appropriate narrative. The result can be responses that are fragmented, redundant, or misaligned with user intent.

Chapter 5

Proposed Solution and Implementation

This chapter presents the proposed solution developed to automate constraint generation for ASIC tools using a RAG-based architecture. It introduces the overall system design, the benchmark designs used for evaluation, and the implementation decisions that guided the selection of frameworks, databases, embeddings, and language models. Special attention is given to the RAG pipeline, detailing its prompting strategies and parameter tuning, which were crucial for adapting the system to tool-specific documentation and improving output reliability.

5.1 Solution Overview

Figure 5.1 illustrates the pipeline implemented to enhance the quality and relevance of responses generated by the LLM using RAG and to evaluate the LLMs' performance. In the initial phase, an embedding model processes both documents and incoming queries to generate embeddings. These embeddings are then stored and indexed in vector databases for efficient similarity searches; this ensures customized retrieval contexts from tool-specific documentation. Upon receiving a query, it is decomposed into multiple logical sub-queries, each retrieving the top-k most relevant document fragments. These fragments are subsequently refined by a reranker into the top-n most pertinent segments, which are then concatenated with the prompt and provided to the LLM.

Within the LLM component, Few-Shot Learning provides illustrative examples guiding the model's reasoning process through CoT prompting. The constraints generated by this process are then evaluated against the reference constraints using similarity and correctness evaluators. This study focus on varying the RAG parameters (top-k and top-n), the prompt, and the LLM selection to comprehensively evaluate system performance.

This pipeline is designed for generating constraints tailored to specific tools and can be scaled for various tool requirements. Although a unified system was considered to integrate constraint generation across tools, the distinct syntax requirements of each tool necessitated maintaining separate vector databases for their respective manuals.

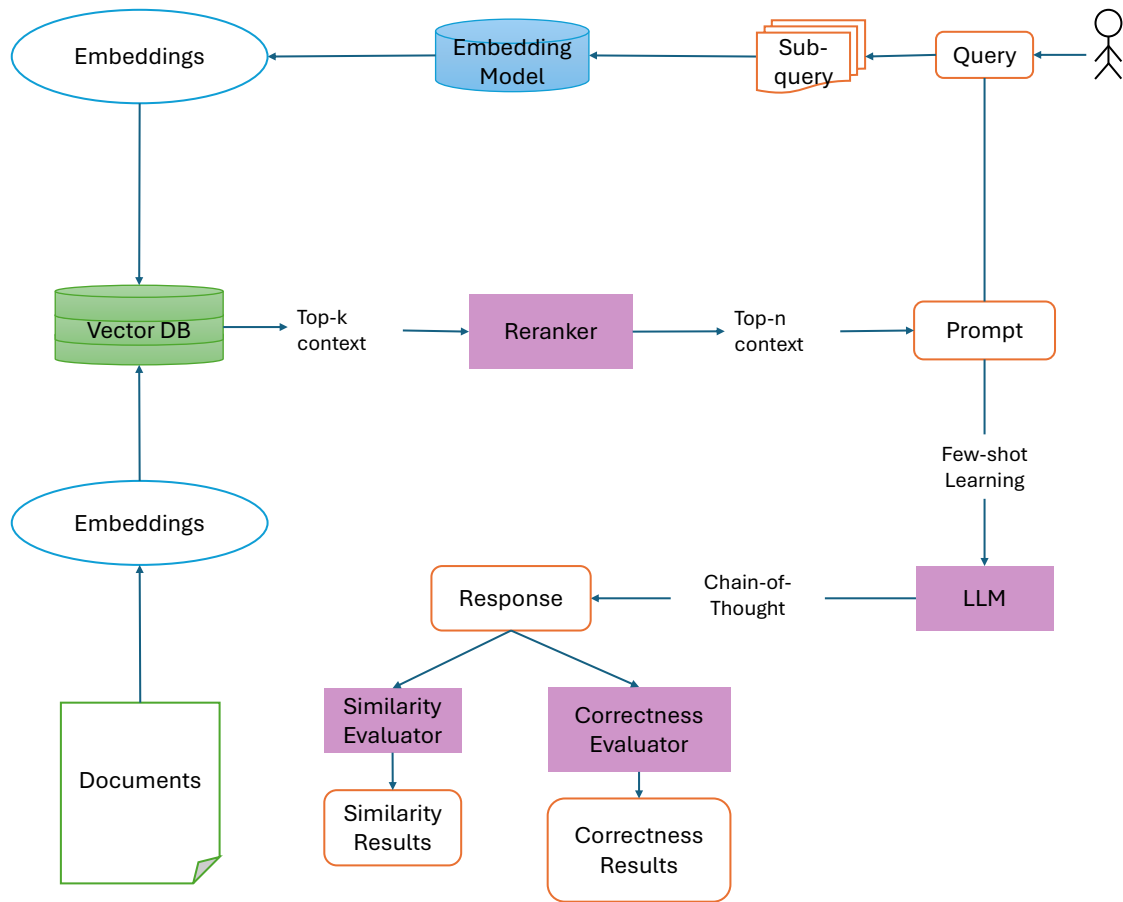


Figure 5.1: System's pipeline.

5.1.0.1 Prompting Strategies

The design of prompting strategies was critical for maximizing the potential of LLMs. Two main strategies, CoT prompting and Few-Shot Learning, were adopted to fulfill the task's specific requirements.

The CoT prompting method was utilized to stimulate reasoning abilities in the models. This approach facilitates the generation of intermediate reasoning steps, assisting the model in decomposing complex problems into smaller, manageable components. Research such as [21] highlights CoT's efficacy in enhancing performance on multi-step tasks and this technique proved to be valuable for tasks requiring refined understanding, which could be beneficial for the generation of design constraints since a detailed comprehension of the design is required.

In parallel, Few-Shot Learning complemented CoT by presenting a limited number of task-specific examples to the model. These examples clearly illustrated the expected reasoning process, enabling the model to generalize relevant patterns and structures for design constraints. The integration of these strategies ensured structured, articulated reasoning outputs from the models.

5.2 Benchmark Design Development

The benchmark designs used in this research were selected with the primary goal of supporting the development and evaluation of the proposed constraint generation system. These were sourced from educational and reference platforms such as VLSI Verify and ASIC World and subsequently validated by Synopsys engineers. Each design followed a defined development flow, in which a complete set of constraints was manually created for two key tools: VC SpyGlass [56] and Design Compiler [57]. VC SpyGlass performs static sign-off checks at the RTL stage, detecting issues such as CDC and RDC early in the design cycle, while Design Compiler synthesizes RTL into an optimized gate-level netlist, meeting timing, area, and power targets. The constraints defined for these tools formed the reference constraints against which the RAG-based system was evaluated.

5.2.1 Design Selection and Motivation

The benchmark designs include an asynchronous FIFO (First In, First Out), a synchronous FIFO, a dual-port RAM, single-port RAM, and a clock divider. This selection aims to cover a variety of common architectural components found in digital systems, specifically those involving memory interfaces, data synchronization, and clocking strategies.

The asynchronous FIFO was included in the benchmark due to its relevance in scenarios involving multiple clock domains. This type of design operates with two independent clock signals, typically a write and a read clock, which introduces specific challenges related to CDCs and RDCs. These characteristics make the asynchronous FIFO particularly suitable for evaluating both the robustness of timing constraint generation and the effectiveness of structural analysis tools. Its inclusion facilitated the examination of CDC and RDC verification using VC SpyGlass and supported the definition of appropriate synchronization-related constraints.

The synchronous FIFO was included in the benchmark as an intentional counterpart to the asynchronous FIFO. Unlike its asynchronous counterpart, the synchronous FIFO operates entirely within a single clock domain, thereby avoiding complexities associated with clock domain crossings. This contrast enabled direct comparisons between two structurally similar designs with significantly different synchronization characteristics. Including both versions allowed assessment of how the system manages designs with and without multi-clock interactions, providing a simpler reference point for evaluating constraint behavior.

The dual-port RAM was selected for its structural and timing complexity, which makes it a valuable component within the benchmark. The design includes two independent clock domains, each associated with its own input and output port, allowing for simultaneous read and write operations. This architectural feature introduces challenges related to data integrity, CDC and parallel access management.

The single-port RAM was included as the benchmark as the counterpart to the dual-port RAM. Operating with a single clock domain and shared read/write interface. Similar to how the synchronous FIFO complements the asynchronous FIFO, the single-port RAM provides a baseline

against which the behavior of more complex memory architecture can be compared. This contrast supports the assessment of how well the system adapts to different levels of architectural complexity.

The clock-divider is a relevant module, as it is used to test constraints related to generated and derived clocks, which are common in digital systems with multiple clock domains. This design takes a high-frequency input clock and produces a lower-frequency clock. From the constraint generation perspective, the clock divider introduces the need to correctly define generated clock relationships, including parameters such as division ratio, source pin, and phase alignment.

5.2.2 Development Flow

Although the primary purpose of the benchmark designs was to evaluate the constraint generation capabilities of the developed system, the design phase was critical to the overall project. Beyond serving as test cases, these designs provided valuable hands-on experience with the digital design flow and industry-standard tools from the Synopsys suite. Thus, this stage extended beyond RTL functional verification and static code analysis (also known as linting) as it further involved identifying and resolving CDC and RDC issues, ensuring that the designs could be successfully synthesized.

The benchmark development process closely followed a standard digital design flow, combining the steps and tools used into an integrated sequence. It began with the construction of verification test benches and RTL compilation using Synopsys Verilog Compiler Simulator (VCS), followed by linting and structural analysis for syntax, CDCs, and RDCs through VC SpyGlass after defining the necessary VC SpyGlass constraints for that particular design. Next, synthesis and constraint definition for the top-level module were carried out using Synopsys Design Compiler, with formal equivalence checking performed by Synopsys Formality to ensure functional consistency. Finally, designs were validated using Synopsys STA tool, PrimeTime to confirm timing integrity and performance. This comprehensive toolchain guaranteed the correctness and readiness of the benchmark designs before their use in constraint evaluation.

The sequence of the steps followed is illustrated in Figure 5.2, visually summarizing the benchmark development process described above.

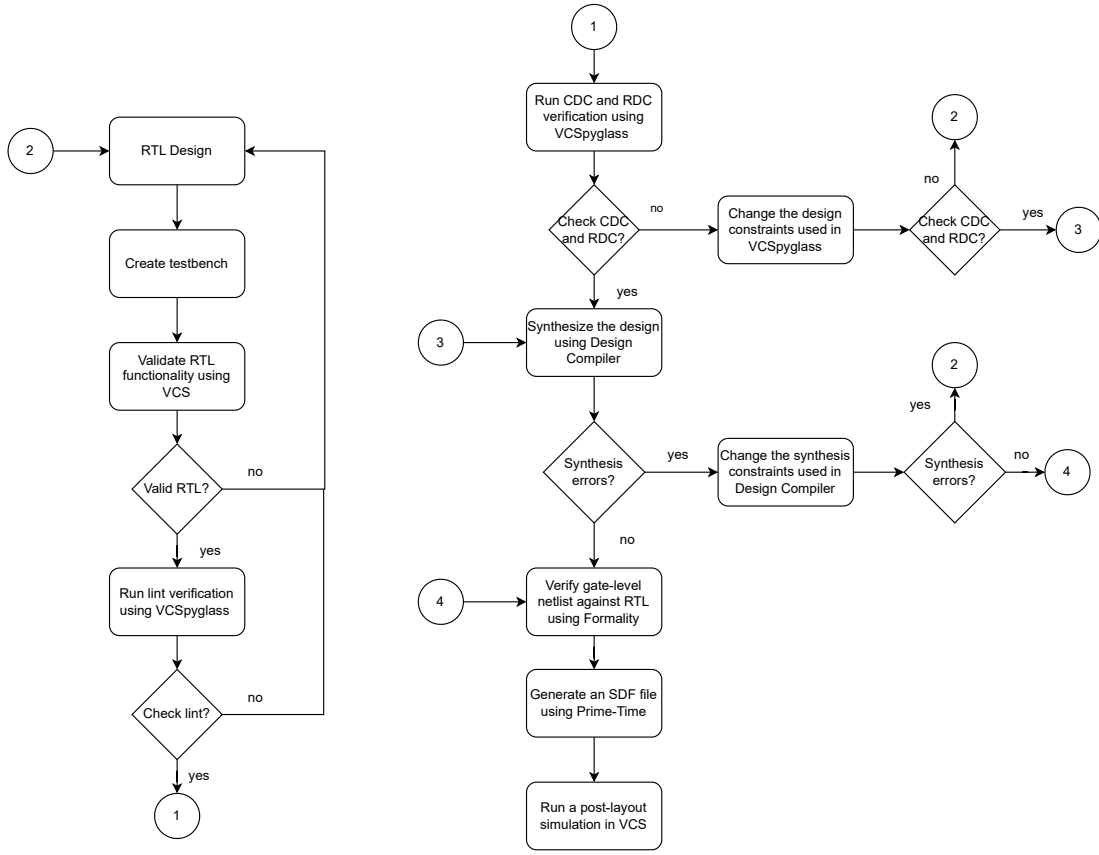


Figure 5.2: Digital design flow followed for benchmark development.

Some standard stages of the digital design flow were intentionally omitted, as they were not considered to contribute meaningfully to the specific objectives of this project. One such stage is scan insertion, a process typically applied during the Design For Testability (DFT) phase to enhance the design's testability by adding scan chains and special register paths that enable controllability and observability of internal flip-flops. While crucial in preparing a design for silicon fabrication and post-silicon validation, this step concerns producing-level fault detection. It does not impact the design's functional correctness, synthesis constraints, or logical structure in a way that could benefit this research. Therefore, it was excluded to maintain focus on the aspects of the flow that directly support constraint generation.

5.2.3 Design Specifications

In this work, the specification chosen to characterize the benchmark designs was the maximum clock frequency, as obtained from the synthesis results. The focus was deliberately placed on the temporal requirements of the designs, particularly on aspects such as CDC and RDC, which are critical for ensuring correct behavior in multi-clock and asynchronous environments.

Table 5.1 summarizes the target clock frequencies, alongside the maximum achievable frequencies.

Table 5.1: Design specifications: target vs. maximum achievable frequencies.

Design	Target max. frequency (MHz)	Max. Achievable (MHz)
Asynchronous FIFO	100	120.57
Synchronous FIFO	50	57.95
Clock Divider	100	102.3
Dual-port RAM	100	119.46
Single-port RAM	100	119.23

During synthesis, the target clock frequency was specified by a clock constraint that the tool aimed to meet. As shown in Table 5.1, the synthesis tool was able to optimize all designs beyond their target frequencies, producing implementations capable of operating at higher rates. This outcome reflects the fact that the benchmark designs are relatively simple and well-structured, allowing the synthesis process to minimize the critical path delays effectively. The maximum frequency achievable corresponds to the inverse of the critical path delay of the synthesized netlist, and its improvement beyond the specified target demonstrates that the designs contain additional timing margin.

5.3 Implementation Decisions

The implementation decisions presented in this section were sustained by the theoretical understanding of RAG gained during the initial research of this dissertation. In particular, Chapter 4 provides the foundations of RAG and optimization techniques, which are especially relevant to the developed system. They offered valuable insights into the importance of selecting an appropriate vector database, choosing suitable embedding for retrieving and a satisfactory language model for generation. A fundamental decision was the selection of the framework to support the construction of the RAG pipeline. The decisions based on the theoretical foundation established, reinforced by the importance of choosing a framework that aligns with the project’s objectives, compose the core selections of the system.

5.3.1 Framework

Several open-source frameworks have emerged to support the development of RAG pipelines. Among the most widely adopted are LangChain [58], Haystack [59], and LlamaIndex [60]; each provides tools to bridge the gap between unstructured data and LLMs, but they differ significantly in usability and architecture.

LangChain is one of the most feature-rich and widely adopted frameworks in the LLM ecosystem. It offers a modular and composable architecture supporting agent-based workflows, reasoning chains, tool integration, and memory management. This framework is particularly well suited for complex applications that require orchestrating multiple components, such as tool-using agents,

retrieval, API calls, and dynamic prompts. However, this complexity comes with higher difficulty and heavier overload.

Haystack is an established framework originally designed for neural information retrieval and question answering. It offers robust support for pipeline configuration, model serving, document storage, and Elasticsearch [61] or Facebook AI Similarity Search (FAISS) [62] back-ends. Elasticsearch is a distributed search and analytics engine widely used for indexing and querying large volumes of text data, while FAISS is a high-performance library for efficient similarity search and clustering of dense vectors. Haystack also includes powerful features for evaluation, filtering, and real-time document updates. Nevertheless, Haystack tends to be heavier in infrastructure requirements and more oriented toward enterprise search applications. For lighter or research-oriented development, it may be more than necessary.

LlamaIndex is a lightweight, high-level framework focused especially on RAG pipelines. It provides tools for document parsing, chunking, indexing, and retrieval, and integrates seamlessly with many embedding models, vector stores, and LLMs. One of the key advantages of LlamaIndex is its simplicity and ease of use, allowing for faster development without excessive code. It is also optimized for performance and is frequently cited as having a quicker and more accurate indexing engine than LangChain [63].

Considering the scope of this research, LlamaIndex was selected as the most suitable framework for implementing the RAG pipeline. In addition, LlamaIndex is already in use within Synopsys, which ensures access to internal support. Compared to the other choices, it offers a more lightweight, focused, and developer-friendly solution for RAG implementation without unnecessary complexity. Furthermore, it benefits from broader community support and extensive documentation, which proved valuable during the implementation and debugging phases.

5.3.2 Vector Database

This system uses ChromaDB as the vector database backend for indexing and semantically retrieving documents. This selection was based on comparative analysis of the various vector stores supported by LlamaIndex, including FAISS, Pinecone, Qdrant, Milvus, Weaviate, and others. The decision considered factors such as performance, ease of integration, infrastructure requirements, and compatibility with the development environment. The selected vector database has the following advantages:

1. **In-memory storage and local execution:** Unlike cloud-based or server-heavy options such as Pinecone and Weaviate, ChromaDB supports local deployment with in-memory storage. By leveraging in-memory storage mechanisms, ChromaDB ensures swift access to data without the latency associated with disk-based systems. Additionally, local execution offers important advantage in terms of data privacy, ensuring that sensitive documentation remain within Synopsys's secured environment.

2. **Simple Application Protocol Interface (API):** The simplicity of the ChromaDB API streamlines the development process by providing developers with a straightforward interface to interact with the database.
3. **Feature-rich collections and querying:** ChromaDB provides a robust and flexible system for managing data through its collections. Each collection can store embeddings along with associated metadata and documents, enabling structured organization of information. Its querying capabilities go beyond basic vector similarity search: developers can filter results by metadata, combine multiple query conditions, and retrieve associated documents or IDs in a single call. This rich feature allows fine-grained retrieval workflows without adding complexity to the overall implementation.
4. **Cost:** ChromaDB is fully open-source, with no usage fees, as opposed to commercial solutions like Pinecone.
5. **Community-driven:** Fast iteration from open source contributors.

For LLM applications, ChromaDB stands out due to its specialized design, tailored for semantic search and natural language processing tasks.

5.3.3 Embedding Models

Throughout the development of the constraints generation system the demand of two different embeddings models emerged, one for indexing and retrieving relevant context from the vector databases and another for the semantic similarity evaluator.

The `mxbai-embed-large-v1` [64] embedding model has a strong performance across a wide range of tasks, as evidenced by its top ranking in the Massive Text Embedding Benchmark (MTEB). This model offers a balance between computational efficiency and semantic accuracy, making it well-suited for complex applications such as RAG. In the context of this dissertation, it was employed to retrieve design constraints from technical documentation. Design constraints are often expressed in structures (command-like language), requiring the embedding model to accurately capture both syntactic structure and semantic intent. The `mxbai-embed-large-v1` model is well-suited to this purpose, as it supports an input of 512 tokens and exhibits robust generalization across natural language. Additionally, it is a light open-source model, facilitating experimentation and local deployment.

However, the RAG outputs were extended passages and required a bigger context length to be properly evaluated. To handle the semantic similarity evaluation, the `nomic-embed-text-v1` [65] embedding model was selected due to its demonstrated excellence across long-context benchmarks. On the MTEB, it surpasses leading closed-source models such as OpenAI's `text-embedding-ada-002`, evidencing its ability to capture nuanced semantic relationship across diverse tasks. Moreover, it achieves top performance in Long-Context (LoCo) benchmark and performs competitively on the Jina Long-Context Benchmark, correctly handling extended passages [66]. These

factors, combined with its open-source availability, render it particularly well-suited for semantic similarity assessment.

5.3.4 Language Models

This subsection comprehensively analyzes the language models employed for constraint generation. Their performance characteristics, strengths and limitations will be discussed by means of establishing the ground for their selection. For this research, models with different parameter values were selected to assess the ideal number for the system deployed. Additionally, both instruction-tuned models, designed to follow explicit prompts, and code-specific models, pre-trained on structured and syntax-driven corpora, were evaluated to assess their respective abilities to generate precise, directive-compliant design constraints.

5.3.4.1 Llama 3.1 8B Instruct

Llama 3.1 8B Instruct represents Meta’s instruction-tuned variant of the Llama 3.1 family, optimized for following instructions to complete a task. The model features 8B parameters and utilizes an auto-regressive transformer architecture with enhanced multilingual capabilities. The 8B parameter size makes it highly accessible for research environments; however, the relatively small parameter count may limit performance on the constraint generation system, since it requires a deep understanding. Additionally, this model has an Intelligence Index of 24, suggesting limitations in complex reasoning tasks. The following table shows the benchmarks for different categories:

Table 5.2: Benchmark Scores of Llama 3.1 8B Instruct (from [67]).

Category	Benchmark	# Shots	Score
General	MMLU	5	69.4%
	MMLU (CoT)	0	73.0%
	IFEval	0	80.4%
Reasoning	ARC-c	0	83.4%
	GPQA	0	30.4%
Code	HumanEval	0	72.3%
Math	GSM-8K (CoT)	8	84.5%
	MATH (CoT)	0	51.9%

5.3.4.2 Gemma 3 Instruct

Gemma-3-27B-IT is Google’s instruction-tuned model, featuring 27 B parameters and multimodal capabilities. It is designed for single-GPU deployment while maintaining competitive performance against significantly larger models, achieving remarkable results that compete with Llama-405B.

However, its output speed of 17.3 tokens per second is slower than average, which may impact real-time applications. Despite this, the Gemma-3-27B demonstrates superior reasoning abilities, with an Intelligence Index of 38, significantly outperforming many larger models and establishing itself as a highly efficient choice for research applications requiring strong reasoning capabilities. Table 5.3 provides an overview of the benchmark performance of Gemma-3-27B.

Table 5.3: Benchmark scores for Gemma 3 27B (from [68]).

Category	Benchmark	# Shots	Gemma 3 27B
General	MMLU	5	78.6%
	MMLU (Pro CoT)	5	52.2%
Reasoning	ARC-c	25	70.6%
	GPQA	5	50.0%
Code	HumanEval	0	48.8%
Math	GSM-8K	8	82.6%
	MATH	4	50.0%

5.3.4.3 Mistral Small 3.1 24B Instruct

Building upon Mistral Small 3, Mistral Small 3.1 adds state-of-the-art vision understanding and enhances long context capabilities up to 128 k tokens without compromising text performance. With 24B parameters, this model achieves top-tier capabilities in text. It is ideal for low-latency function calling, local inference for hobbyists and organizations handling sensitive data, as well as advanced tasks such as long document understanding, complex programming, and mathematical reasoning, making it a versatile choice for both research and production environments. The respective benchmarks are presented in the following table:

Table 5.4: Benchmark scores for Mistral-small 3.1 24B Instruct (from [69]).

Category	Benchmark	# Shots	Scores
General	MMLU	0	80.62%
	MMLU (CoT)	5	66.76%
Reasoning	GPQA	0	44.42%
Math	MATH	0	69.3%

5.3.4.4 Llama 3.3 70B Instruct

Llama-3.3-70B-Instruct represents a significant scaling of the Llama architecture to 70B parameters, offering enhanced capabilities for complex reasoning and instruction-following tasks. The

70B parameter count provides substantial improvements in reasoning and comprehension; however, it requires significant GPU memory and computational resources and the large parameter count results in slower inference times. As expected from a larger model, Llama 3.3 demonstrates strong performance across benchmarks, as presented in Table 5.5.

Table 5.5: Benchmark scores for Llama3.3-70B-Instruct (from [70]).

Category	Benchmark	# Shots	Score
General	MMLU	5	83.6%
	MMLU (CoT)	0	86.0%
	IFEval	0	87.5%
Reasoning	ARC-c	0	94.8%
	GPQA	0	46.7%
Code	HumanEval	0	80.5%
Math	GSM-8K (CoT)	8	95.1%
	MATH (CoT)	0	68.0%

5.3.4.5 DeepSeek-R1-Distill-Llama-70B

DeepSeek-R1-Distill-Llama-70B represents an innovative approach to model development, distilling the reasoning capabilities of larger models into a more efficient 70B-parameter architecture while maintaining competitive performance. It combines the reasoning patterns of larger models with improved efficiency, particularly excelling on math, code, and reasoning benchmarks.

Its output speed of 64.3 tokens per second, however, is below average, which limits its suitability for real-time applications. Moreover, while it excels at reasoning tasks, it may not deliver equally strong performance across all domains, as shown in Table 5.6. Nonetheless, DeepSeek-R1-Distill-Llama-70B could be an excellent choice for research and applications that demand advanced analytical and problem-solving capabilities.

Table 5.6: Benchmark scores for DeepSeek-R1 distill Llama 70B (from [71]).

Category	Benchmark	# Shots	Score
Reasoning	GPQA Diamond	1	65.2%
Code	LiveCodeBench	1	57.5%
Math	AIME 2024	1	70.0%
	MATH 500	1	94.5%

5.3.4.6 Qwen2.5 Coder 14B and 32B

The Qwen2.5-Coder series represents Alibaba Cloud’s latest code-specific large language models, succeeding the CodeQwen1.5 series. These instruction-tuned models are designed for coding tasks while maintaining strong mathematics and general competencies performance. The flagship 32B parameter achieves state-of-the-art performance among open-source code LLMs, with coding abilities comparable to GPT-4o. The 14B parameter variant offers a balanced approach between performance and efficiency. While maintaining the same transformer architecture and training methodology as its larger counterpart, it provides faster inference and lower resource requirements. The following table shows the benchmarks for both models:

Category	Benchmark	# Shots	Qwen2.5 Coder 14B	Qwen2.5 Coder 32B
General	MMLU	0	75.2%	79.1%
Reasoning	ARC-c	0	70.5%	66.0%
Code	HumanEval	0	92.7%	89.6%
	LiveCodeBench	0	31.4%	23.4%
Math	MATH	4	57.2%	52.8%
	GSM-8K	4	91.1%	88.7%

Table 5.7: Benchmark scores for Qwen2.5 Coder 14B and Qwen2.5 Coder 32B (from [72]).

5.4 Similarity and LLM-based Scores

Accuracy is the primary measure of the model’s effectiveness in handling complex reasoning and factual recall for constraint generation. Similarity and LLM-based scores were used to determine whether the generated constraints aligned with the expected ones. The evaluation strategies for this study were designed to ensure accurate and consistent assessment of the model’s performance to generate constraints. The evaluation required a detailed approach to capture semantic meaning and reasoning behind the model-generated constraints.

A cross-encoder-based semantic similarity metric from the LlamaIndex framework was used [73], with a similarity cut-off of 0.8 (a fixed value by LlamaIndex) to decide whether a predicted constraint was sufficiently close in meaning to its expected counterpart. Predictions scoring at or above this cut-off were treated as semantically equivalent, accommodating variations in phrasing while preserving overall intent and minimizing false positives.

To address the shortcomings of semantic similarity, LLMs were introduced as evaluators, also imported from LlamaIndex framework. The function uses a prompt that provides the LLM with a user query, a reference answer, and a generated answer [74]. The LLM is instructed to judge the relevance and correctness of the generated answer, output a single score between 1 (worst) and 5 (best), and provide reasoning for the score. The prompt explicitly states: if the answer is not

relevant, score 1; if relevant but contains mistakes, score 2–3; if relevant and fully correct, score 4–5.

Together, these complementary metrics create a robust quantitative evaluation of the system's output. Semantic similarity provides an objective measure of alignment between generated and reference constraints, while the LLM-based scoring introduces a layer of contextual and reasoning-aware judgment.

5.5 RAG Implementation and Parametrization

This section presents the RAG implementation and parameter variations, outlining how retrieval and prompting strategies were designed and adjusted to optimize constraint generation within the system.

5.5.1 RAG Implementation

RAG plays an essential role in this research by ensuring the model's outputs remain precise, contextually accurate, and consistent with the domain-specific knowledge crucial for generating constraints. The implemented prompt, shown in Figures 5.3 and 5.4, was meticulously crafted to steer the LLM's reasoning, guaranteeing alignment with the supplied context and enhancing the interpretability of its responses. The prompt addresses several key objectives:

1. **Role Definition:** The phrase "As a Digital Design Engineer" explicitly sets the model's perspective, ensuring its responses meet engineering expectations [75].
2. **Context Limitation:** The clear directive "Only use information from the context" confines the model's responses to the retrieved data, reinforcing it to follow guidelines.
3. **Format Constraint:** The instruction "Don't include any additional information or explanations, just the constraints" strictly regulates the output format of the LLM.
4. **Input/Outputs:** The input/output names of the benchmark designs were included since in the initial trials some of the models were hallucinating. This ensures the models digest well their names for constraint generation.
5. **Multilingual Functionality:** Mandates responses to be delivered in the same language as the query, thus broadening usability across various linguistic environments.

The context provided within the prompt is dynamically generated using a retrieval mechanism that indexes and queries carefully chosen tool manuals. This procedure ensures that the most relevant information is supplied to the model through these precise steps:

1. **Document selection:** The document selection was carried out through a careful and deliberate review process. The process involved searching through the company's internal

databases to locate official documentation for the tools under study. From this initial collection, the manuals and guides directly addressing design constraints were identified and extracted, forming a curated set of documents to serve as the base for the system's knowledge base.

2. **Multi Query Decomposition:** The nature of design constraining is inherently complex and typically involves addressing multiple aspects, and cannot be captured in a single query. The query decomposition allows focused sub-queries, each targeting a specific task that is then mapped to a retrieval step, extracting the most relevant context from the indexed database.
3. **Context Injection:** Retrieval fragments are inserted into the context of the prompt, forming the basis for the model's response generation.

This retrieval step equips the generative model with targeted, domain-specific knowledge, effectively bridging static information and practical applicability.

```
Context:
{context}
-----
Project:
{design}

Brief Description of the Design:
{description}

Input ports of Design:
{inputs}

Output ports of Design:
{outputs}

Verilog Code of the Design:
{code}
-----
Task:
Analyze the project contents to:

1. Identify all clock and reset signals (names, active-high or active-low).
2. Use the provided input/output ports to generate constraints, do not infer
any additional ports from the Verilog.
3. Detect any synchronizers.
-----

You're a Digital Design Engineer tasked to create the design constraints for
the project above.
Only use information based on the context and provide the design constraints
to:

{query}

Don't include any additional information or explanations, just the
constraints.
```

Figure 5.3: Zero-shot CoT prompt designed for the system.

```

Example 1: ASYNCHRONOUS FIFO Design

Description:
{description_a_fifo}

Verilog Code:
{example_a_fifo}

Expected Response:
{expected_constraints_a_fifo}

-----

Example 2: Clock Divider Design

Description:
{description_clk_div}

Verilog Code:
{example_clk_div}

Expected Response:
{expected_constraints_clk_div}
-----

Context:
{context}
-----

Project:
{design}

Brief Description of the Design:
{description}

Input ports of Design:
{inputs}

Output ports of Design:
{outputs}

Verilog Code of the Design:
{code}
-----

Task:
Analyze the project contents to:

1. Identify all clock and reset signals (names, active-high or active-low).
2. Use the provided input/output ports to generate constraints, do not infer any
   additional ports from the Verilog.
3. Detect any synchronizers.

You're a Digital Design Engineer tasked to create the design constraints for the project
above.
Only use information based on the context and provide the design constraints to:

{query}

Ensure the response follows the reasoning demonstrated in the example 1 (asynchronous
FIFO design) and example 2 (Clock Divider design).
Don't include any additional information or explanations, just the constraints.

```

Figure 5.4: Few-shot CoT prompt designed for the system.

5.5.2 RAG Parameters Variations

The RAG approach uses two parameters, the number of documents chunks retrieved from the underlying knowledge (top-k search) and the number of documents chunks re-ranked for relevance and passed to the generation phase (top-n rerank). In this study, we explored a range of values parameters, varying top-k search from 5 to 20 and top-n rerank from 3 to 10, to better understand their impact on model performance and optimize their configuration for design constraining.

Increasing the number of retrieved document chunks expands the breadth of contextual information available, providing the model with a more comprehensive basis for constraint generation. However, retrieving excessive chunks may introduce irrelevant or noisy data, potentially reducing the model's ability to focus effectively. The evaluation aimed to determine an optimal balance between retrieval depth and information relevance, ensuring sufficient context without sacrificing quality.

Determining how many retrieval chunks are re-ranked and subsequently used in the generation phase is vital for filtering out less pertinent information and prioritizing the most relevant context. Smaller values enhance precision but might exclude beneficial details, while larger values provide a richer context but can dilute the relevance if noise from the retrieval stage is present. By varying the top-n rerank, this research evaluated how reranking depth influences model accuracy and reasoning capabilities.

Furthermore, the parameter exploration provided insights into computational trade-offs associated with these parameters. These findings are presented in the next chapter, where the overall impact of the chosen configurations is examined in detail.

Chapter 6

Results and Evaluation of RAG-based prototype

To validate the generated constraints, a four-part evaluation methodology was developed: a quantitative analysis, an inference time analysis, a tool-based inspection, and an expert review. This chapter presents the findings derived from each of these evaluation stages. For the tool inspection and expert review, two representative models were selected to illustrate the methodology in depth. This evaluation method aims to demonstrate not only the quantitative performance of the system but also the ability to efficiently produce meaningful, usable constraints in realistic verification and synthesis contexts.

6.1 Quantitative Evaluation

This section reports the quantitative results for every model, prompt, and retrieval configuration. Two complementary visualizations convey the data distribution: box plots and heatmaps. The similarity score assesses the semantic alignment between the generated constraints and the reference and is scaled from 0 (complete mismatch) to 1 (perfect match).

The correctness score measures logic validity and ranges from 0 (invalid) to 5 (entirely correct). Box plots summarize each metric’s central tendency and spread for a single factor at a time. Simultaneously, the heat maps integrate all three factors, highlighting the best and worst performing triads. Together, these visual tools provide granular and holistic insight into model behavior.

The box plot distributions present in Fig. 6.1 and Fig. 6.2 allow for a evaluation of how different configurations influence the quality of the output generated. As highlighted, the configuration ‘top_k=5, top_n=3’ and ‘top_k=10, top_n=5’ reach higher median for similarity scores across models and prompting strategies while keeping median correctness fixed for all configurations.

In contrast, the broader retrieval window, ‘top_k = 20, top_n = 10’ produced a lower medians. The larger number of retrieved chunks diluted the relevance of the context and increased

the cognitive load on the model, suggesting that the selected models do not handle long context efficiently.

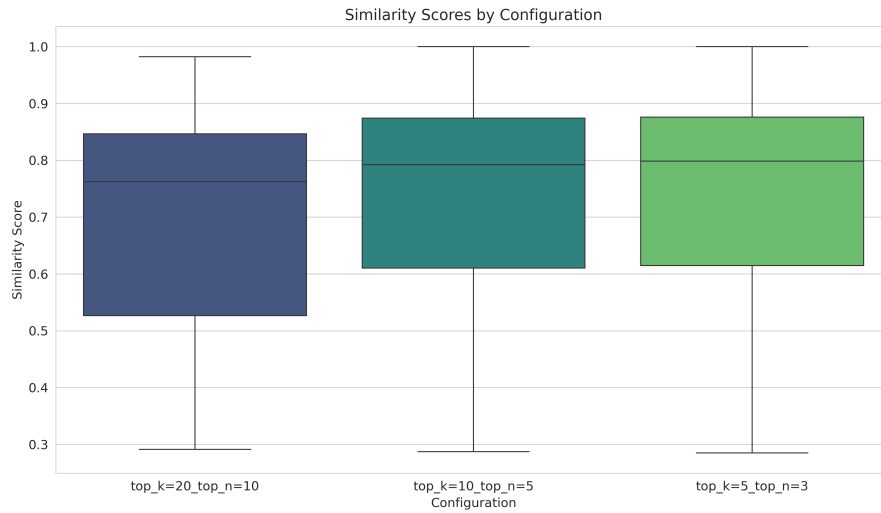


Figure 6.1: Box plot of similarity scores by configuration.

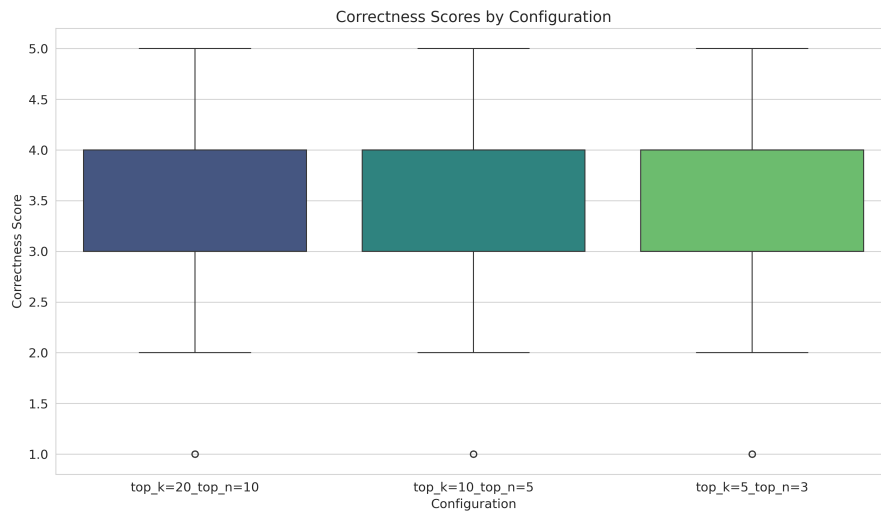


Figure 6.2: Box plot of correctness scores by configuration.

When comparing the two prompting strategies, few-shot prompting achieved a higher median similarity score maintaining the correctness scores fixed, as shown in Fig. 6.3 and Fig. 6.4. This improvement can be attributed to the presence of explicit examples in the prompt, which guided the model toward the expected constraint format. Conversely, in the zero-shot setting, models had to infer both the task and the desired output structure solely from the description and code, which proved more challenging and led to less consistent outputs.

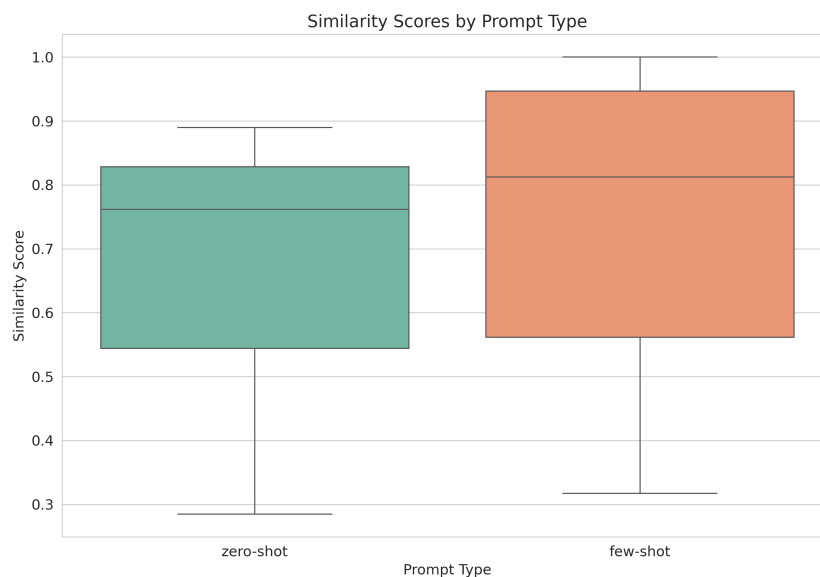


Figure 6.3: Box plot of similarity scores by prompt strategy.

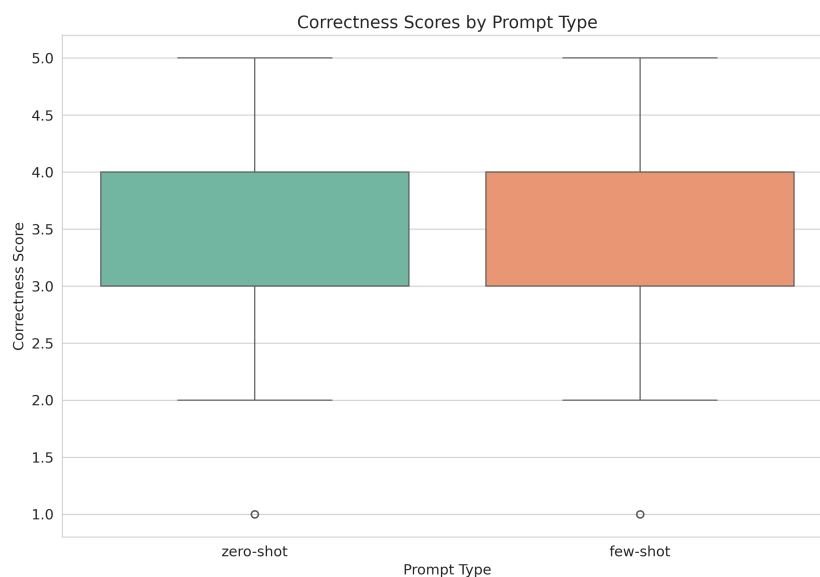


Figure 6.4: Box plot of correctness scores by prompt strategy.

The box plot distributions in Fig. 6.5 and Fig. 6.6 reveal two distinct performance ranks. Llama 3.3 70B stands out: it posts the highest median similarity and its lower quartile sits above the similarity threshold (0.8). At the same time, its correctness maintains a consistent score, with a cluster around 4. Gemma 3 27B, Mistral Small, Llama 3 8B form a second, but still strong group, sharing similarity medians near the similarity threshold and around 3 and 4 for correctness, indicating consistently relevant and mostly accurate constraint sets. By contrast, the two Qwen-Code models lag markedly behind. Qwen-Code-14B shows a wide spread: although its upper quartile touches the stronger models, its median similarity drops to about 0.75 and its correctness

plunges to 1, signalling frequent invalid outputs. Qwen-Code-32B is the clear outlier: its similarity median is near 0.4, and every correctness observation centers on 1, pointing to systematic failures in relevance and validity.

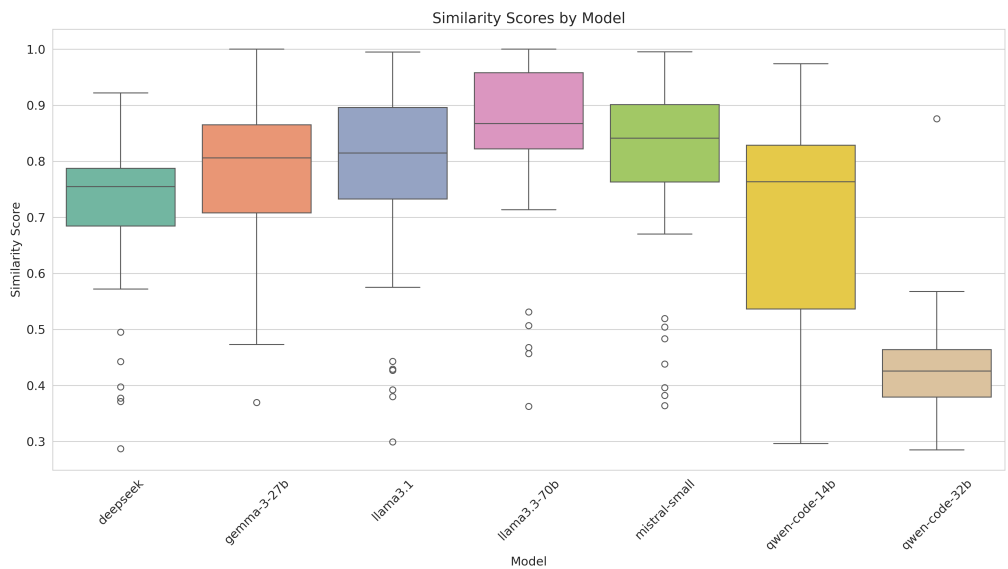


Figure 6.5: Box plot of similarity scores by model.

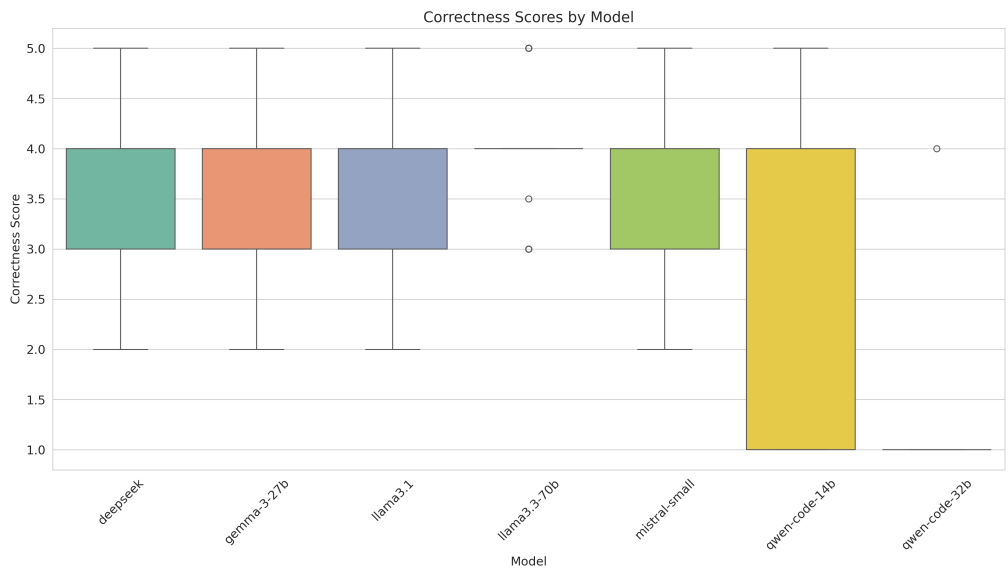


Figure 6.6: Box of for correctness scores by model.

Figure 6.7 integrates the prior analyzes of retrieval configuration, prompting strategy, and model performance into a single heatmap. The graphic follows a 2×2 matrix structure: the upper row portrays few-shot results, whereas the lower row depicts zero-shot results. Within each row, the left-hand panel presents the mean similarity scores and the right-hand panel the mean correctness scores. The abscissa of every panel enumerates the seven models under evaluation, while

the ordinate lists the three retrieval settings (top_k, top_n). Each cell is labelled with its numerical value and shaded along a continuous colour scale: lighter hues denote superior performance, while darker hues are inferior. By consolidating all factors into a single visual, the heat map furnishes a succinct yet comprehensive perspective on the interaction of model, prompt, and retrieval configuration, thereby facilitating swift identification of the most and least effective combinations.

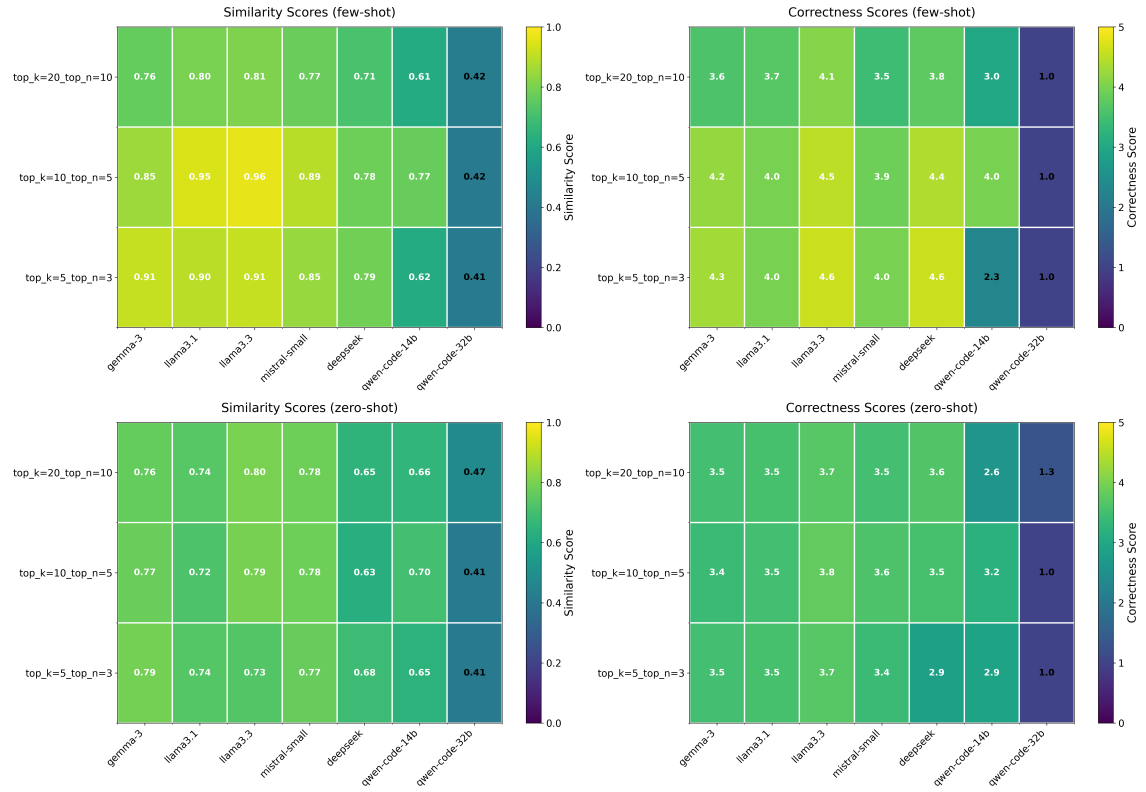


Figure 6.7: Heatmap of final results.

Overall, the similarity scores tended to be in the similarity threshold, reflecting the challenge of producing outputs that match the reference constraints. As for the correctness scores, the results for most models were satisfactory, indicating that these produced logical constraints; however, it still doesn't ensure that the constraints are synthetically correct. Therefore, combining the two scores is essential to interpret the output generated by the models and examine whether the constraints are valid and usable in practice.

It was also observed that, in several cases, the similarity score successfully captured syntactic errors, including formatting issues, as illustrated by the comparison between the generated constraints and the expected reference constraints in Listings 6.1 and 6.2, respectively. In addition, the score highlighted instances where extra constraints were introduced that, although syntactically valid, were incorrectly applied to the design (Listings 6.3 and 6.4). On a positive note, the models did not generate entirely new constraints on their own, indicating that they were effectively guided by the prompt design and the context provided through the RAG system.

```
1 set_output_delay 3 -max -clock clk_a [get_ports "data_out_a"]
```

```

2 set_output_delay 2 -min -clock clk_a [get_ports "data_out_a"]
3
4 set_output_delay 3 -max -clock clk_b [get_ports "data_out_b"]
5 set_output_delay 2 -min -clock clk_b [get_ports "data_out_b"]

```

Listing 6.1: Output timing constraints generated by Llama 3.1 for Dual-port RAM.

```

1 set_output_delay 3 -max -clock clk_a [get_ports {data_out_a[*]}]
2 set_output_delay 2 -min -clock clk_a [get_ports {data_out_a[*]}]
3
4 set_output_delay 3 -max -clock clk_b [get_ports {data_out_b[*]}]
5 set_output_delay 2 -min -clock clk_b [get_ports {data_out_b[*]}]

```

Listing 6.2: Expected output timing constraints for Dual-port RAM.

```

1 create_generated_clock [get_ports "empty"] -source [get_ports "rclk"] -name
2 empty_clk -master_clock "rclk"
3
4 create_generated_clock [get_ports "full"] -source [get_ports "wclk"] -name full_clk
5 -master_clock "wclk"

```

Listing 6.3: Addition of a timing constraint by Mistral Small.

```

1 set_disable_timing -from [get_ports {g_wptr_sync}] -to [get_ports {g_rptr_sync}]
2
3 set_disable_timing -from [get_ports {g_rptr_sync}] -to [get_ports {g_wptr_sync}]

```

Listing 6.4: Addition of a timing constraint by Gemma 3.

In the particular case of DeepSeek the model produced syntactically correct constraints but prefaced them with an excessively long explanatory section. This verbosity reduced the semantic similarity to the concise expected output and the correctness score given by the LLM judge, even though the actual constraints embedded in the response were correct.

It was also observed that Mistral Small generated incomplete set of constraints for VC SpyGlass (Listing 6.5). This behavior may be attributable to the high context used in the prompt. This suggests that this model is more susceptible to prompt misinterpretation and may fail to correctly infer the expected constraints.

```

1 ##SECTION1 : PRIMARY CLOCKS ###
2 create_clock [get_ports "clk"] -name clk -period 10
3
4 ##SECTION 2 : CREATE RESET ###
5
6 create_reset [get_ports "rst_n"] -name rst_n -async -type reset -sense low
7
8 ## SYNCHRONIZERS ##
9 set_asynchrst_synchronizer -cell arest_sync

```

Listing 6.5: VC SpyGlass generated constraints by Mistral Small for FIFO. The model did not specify any inputs and outputs delays.

Additionally, the code-specific models demonstrated poor performance. Although the models were not pre-trained on HDL, their proficiency in generating code could be an asset in generating constraints, however this did not applied. In fact, both code-specific models attained the worst performance for both scores.

6.2 Inference Time Analysis

All experiments were executed on a NVIDIA A100-SXM4 GPU with 80 GB of memory; this setup provided the computational capacity required to handle the largest models under evaluation. Inference time is a critical factor for assessing the practicality of integrating LLMs into an automated constraint generation flow. The results, present in Figures 6.8 and 6.9, show significant variation of inference times across the selected models.

In general, Llama 3.1 was the fastest model, with inference times consistently below 7 seconds across all scenarios. At the opposite end of the spectrum, Qwen-Code-14B exhibited extremely long inference times, particularly under the most demanding configuration, where it reached in average of 167.36 s. Among the mid-range performers, Llama 3.3 70B is particularly noteworthy. Despite being a heavier model, it maintained inference times within a reasonable range, with a average of 39.41 s, while also delivering the highest overall performance in similarity and correctness scores. This balance between quality and efficiency positions Llama 3.3 as a highly promising option.

Gemma-3-27B and Mistral-Small also showed satisfactory inference times, though their constraint-generation quality did not match that of Llama 3.3. In contrast, DeepSeek and Qwen-Code-32B incurred much heavier computational costs, with inference times consistently above one minute in most scenarios, making them less suited for time-sensitive applications.

Qwen-Code-14B was the slowest model, recording a mean inference time of 167.36 s under the broader configuration, roughly double the inference time of the other models. The cause of this behavior is not fully understood.

In summary, the inference time evaluation reinforces the conclusion that Llama 3.3 offers the best trade-off between high-quality constraint generation and practical processing time, standing out as both accurate and efficient.

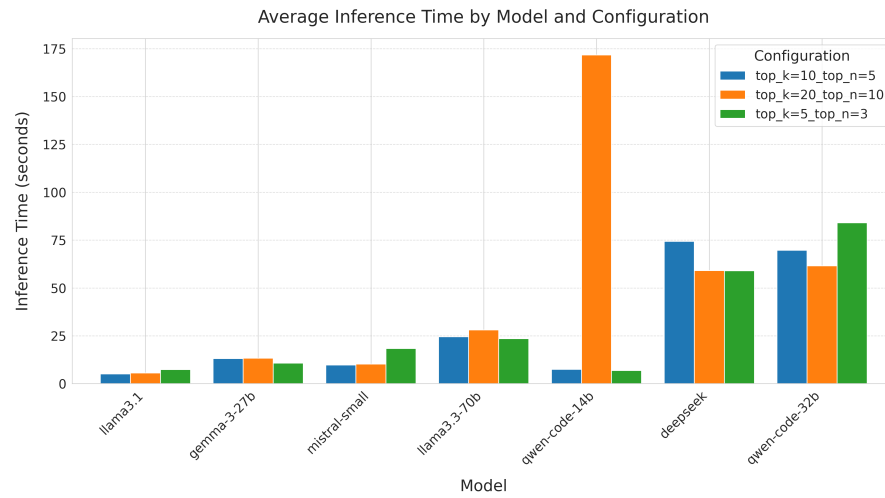


Figure 6.8: Average Inference Time of configurations by model.

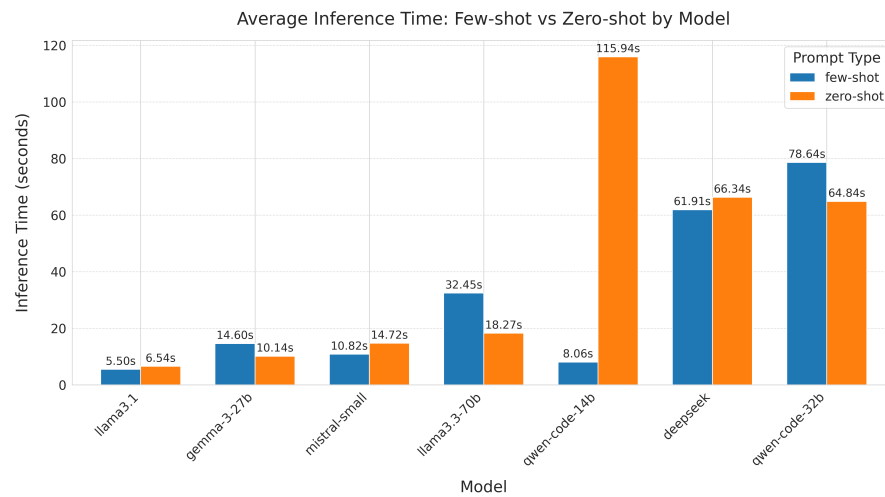


Figure 6.9: Average Inference Time of few-shot and zero-shot prompts by model.

Furthermore, when examining both figures, no consistent trend emerges linking retrieval configurations or prompting strategies to major shifts in inference time. In other words, increasing the amount of retrieved context, for example, moving from narrower configurations to broader ones, did not lead to the slowdowns that might be expected. Similarly, the choice between few-shot and zero-shot prompting showed only minor influence on overall latency. This suggests that the evaluated LLMs are generally capable of handling larger input contexts without a significant penalty in processing time, demonstrating their robustness for workflows that require flexible retrieval depths.

6.3 Tool-based Inspection

Two models with markedly different constraint outputs were deliberately selected to observe how their results would be interpreted by the tools. The models were: Llama 3.3 70B and Mistral Small 24B. The Mistral output included incomplete constraints and substituted statements that deviated from the expected set. This raised two critical questions: could partially specified or imperfect constraints still pass verification, and might LLM-generated constraints at times prove more precise than those crafted manually? The following table summarizes the number of errors and warnings reported by VC SpyGlass across lint, CDC, and RDC analyzes for the constraints generated by Mistral.

	Lint		CDC		RDC	
	Errors	Warnings	Errors	Warnings	Errors	Warnings
Asynchronous FIFO	0	0	5	0	0	0
Synchronous FIFO	0	0	10	1	0	1
Clock Divider	0	0	0	1	0	1
Dual-port RAM	0	0	13	0	0	0
Single-port RAM	0	0	5	1	0	1

Table 6.1: Feedback of constraints generated for VC SpyGlass by Mistral model.

The errors identified in the constraints generated by the Mistral model stem from the incompleteness of its constraint set, most notably, the omission of input and output delay definitions. These timing constraints are critical because they establish the timing relationships between the design's input/output signals and the clock, effectively modeling more demanding scenarios in which the design must operate reliably. Overall, the Mistral Small model generated constraints performed poorly under VC SpyGlass checks, an outcome that contrasts sharply with the results obtained from Llama 3.3, as detailed in Table 6.2.

	Lint		CDC		RDC	
	Errors	Warnings	Errors	Warnings	Errors	Warnings
Asynchronous FIFO	0	0	0	0	0	0
Synchronous FIFO	0	0	0	1	0	1
Clock Divider	0	0	0	1	0	1
Dual-port RAM	0	0	0	0	0	0
Single-port RAM	0	0	0	1	0	1

Table 6.2: Feedback of constraints generated for VC SpyGlass by Llama 3.3 model.

Within the Synopsys workflow, warnings are treated with the same level of attention as errors. They should only remain in the design if their nature is fully understood by the engineer and

deemed harmless for that specific implementation. Following this principle, the clock divider warning can be safely disregarded, as it merely signals that `clk_out` is not propagated through the rest of the design—an acceptable condition given that this block’s sole purpose is to generate an output clock.

Similarly, warnings observed in the Single-Port RAM and Synchronous FIFO designs point to a missing `set_clock_groups` constraint. This constraint is used to define clock domains that are mutually exclusive or asynchronous, ensuring that timing paths between them are excluded from analysis. In these cases, because each design operates under a single clock, the warning is inconsequential and may also be ignored.

In summary, Llama 3.3 produced a complete and well-structured constraint set for VC SpyGlass, resulting in superior performance. This advantage can be attributed, at least in part, to the model’s significantly larger parameter count, which provides greater capacity to interpret prompts and generate precise constraints.

For the synthesis, all constraints generated by Llama 3.3 were successfully parsed by Design Compiler. The same could not be said for Mistral, whose incomplete constraints led to issues during processing, as summarized in Table 6.3.

	Errors	Warnings
Asynchronous FIFO	0	0
Synchronous FIFO	1	1
Clock Divider	0	0
Dual-port RAM	0	6
Single-port RAM	1	0

Table 6.3: Feedback of constraints generated for Design Compiler by Mistral model.

For the Synchronous FIFO, the tool failed to recognize one of the constraints generated by Mistral, resulting in both an error and a warning. In the case of the Dual-Port RAM, warnings arose when defining the relationship between data and clock signals; this issue stemmed from using the `get_ports` command to reference ports directly. Mistral introduced a non-existent constraint for the Single-Port RAM, which the tool could not interpret, leading it to be disregarded entirely during processing.

Returning to the key questions posed at the beginning of this section, the results provide a clear answer. The constraints produced by Mistral demonstrate that incomplete or incorrectly formulated specifications are insufficient for tool validation: missing input and output delays, fabricated statements, and omissions led to systematic warnings, errors, and, in some cases, complete rejection by the tools. In contrast, Llama 3.3 generated a complete and compliant set of constraints that was fully interpreted by both VC SpyGlass and Design Compiler, showing that when an LLM has sufficient capacity and is guided by crafted prompts, it can produce outputs that meet expectations.

These findings underscore a broader conclusion: tool compatibility depends on completeness, precision, and established syntax and semantics adherence. Only models capable of delivering such rigor, like Llama 3.3, can meaningfully support and accelerate real-world verification and synthesis flows. In contrast, models producing partial or inconsistent constraints risk introducing more problems than they solve.

6.4 Expert Review

For the expert evaluation, a questionnaire was distributed to three specialists in the field of ASIC digital design to assess the generated constraints across four key dimensions: functional correctness, completeness, coverage, and adjustment effort. The experts provided numerical ratings (on a 0–5 scale) and free-form comments, detailing the nature of any deficiencies and offering suggestions for improvement. The expected constraints for VC SpyGlass and Design Compiler were provided for each design, along with the constraints generated by two models. The engineers did not know what model they were judging. Model 1 represented Mistral Small and Model 2 represented Llama 3.3. The Table below presents the answers from the experts’ review.

Design		Model 1		Model 2	
		VC SpyGlass	Design Compiler	VC SpyGlass	Design Compiler
Asynchronous FIFO	Functional Correctness	4	3.3	4	4
	Completeness	4.33	3.3	4.33	4.66
	Coverage	4	3.3	4	4.66
	Adjustment Effort	3.66	3.66	4.66	4.66
Synchronous FIFO	Functional Correctness	3.33	3.66	4.66	4.66
	Completeness	3.33	3	4.66	4.66
	Coverage	3	3	4.33	4.33
	Adjustment Effort	2.33	2.33	5	4.66
Dual Port RAM	Functional Correctness	3.66	4.66	4.66	5
	Completeness	3.33	4.66	4.66	5
	Coverage	3.33	4.33	4.33	4.66
	Adjustment Effort	2.33	4.66	4.66	5
Single Port RAM	Functional Correctness	3.33	3.66	4.66	5
	Completeness	3.33	3.66	4.66	5
	Coverage	3.66	3.33	4.33	4.66
	Adjustment Effort	2.33	1.33	4	4.33
Clock Divider	Functional Correctness	4.33	4.33	5	5
	Completeness	4.66	4.66	4.66	4.66
	Coverage	4.33	4.33	4.66	4.66
	Adjustment Effort	4	4.33	4.66	4.33

Table 6.4: Feedback from experts for all designs.

As anticipated, Model 2 (Llama 3.3) achieved the strongest evaluation scores across all criteria. The engineers’ feedback provided valuable insight into the models’ behavior. They noted

that Model 1 (Mistral) frequently misclassified full and empty flags as clock signals, generating incomplete constraints. One comment also observed that, despite these shortcomings, the constraints produced by both models shared a consistent structure and annotation style, indicating that the system could preserve a coherent formatting framework even when the content itself varied in quality.

This expert assessment forms the final component of the evaluation methodology developed in this dissertation, complementing the quantitative analysis, inference time study, and tool-based inspection. Beyond validating the technical reliability of the generated constraints, the expert review provides insight into what is expected in practice by professionals working in ASIC digital design.

Chapter 7

Conclusions and Future Work

This chapter brings the dissertation to an end by revisiting the research questions initially posed and discussing how they were addressed. It then outlines the key contributions of the study, before discussing potential directions for future work, highlighting opportunities for further investigation and development built upon the groundwork laid by this research.

7.1 Research Answers

The study was framed around two research questions that align with the proposed hypothesis. The following paragraphs offer a detailed discussion and reflection on each of these questions individually.

1. RQ1: How effective are LLMs, when combined with Retrieval-Augmented Generation and prompt engineering, in generating accurate and usable design constraints?

The experiments demonstrate that integrating retrieval-augmented generation significantly enhances the ability of large language models to select and generate accurate design constraints. By coupling the models with a curated knowledge base, composed of tool user guides that present, explain, and, in some cases, illustrate constraints with examples, the output becomes more domain-specific and markedly less prone to hallucination. This effect was particularly evident as model size increased, with Llama 3.3 showing the most outstanding resilience against erroneous generations. Furthermore, design constraints are inherently complex: they are command-like statements with diverse syntax, numerous attributes, and a strong dependence on the unique requirements of each design. Because the knowledge base does not encompass this variety, successful constraint generation demands that the LLM demonstrate exceptional reasoning capabilities. Illustrative examples were incorporated directly into the prompts to ease this reasoning process and reduce the likelihood of hallucinations. These examples guided the models toward the expected format for different designs, constraining their syntax and improving consistency. Optimizing the search and rerank parameters further aligned constraint generation with the expected constraint.

2. **RQ2: How can such a tool be seamlessly integrated into existing ASIC design workflows to ensure it is both practical and valuable for engineers?** Although the developed system was not integrated into any ASIC digital design flow, the elaborated research indicates that a valid and rich approach could be to incorporate the tools directly into the system, to integrate post validation. In the context of the prompt, the errors and warnings generated by the tools could be provided to the LLM, allowing it to regenerate or refine the constraints. Such an approach would close the loop between constraint generation and tool verification, making the system progressively brighter with each iteration. This integration would transform the current static generation into a self-correcting, semi-autonomous assistant for ASIC designers, capable of proposing constraints and evolving them dynamically based on honest tool feedback.

7.2 Main Contributions

1. Demonstration of Feasibility

The dissertation provides initial evidence that LLMs when integrated with retrieval-augmented generation pipelines and carefully designed prompts, can closely approximate, and in some instances, replicate the expected constraints. However, the results also reveal important limitations. Smaller models often struggled to produce fully complete or tool-compliant constraint sets. This highlights that, while the approach is promising, its effectiveness depends on model capacity and configuration, suggesting that further refinement and model selection are crucial for consistent performance.

2. Comparative analysis of prompt techniques

A systematic comparison between zero-shot and few-shot prompting techniques is provided, revealing how the inclusion of targeted examples in prompts improves the precision, consistency, and structure of the generated constraints. The findings highlight the importance of prompt engineering in guiding LLM behavior for domain-specific applications.

3. Comparative analysis between models

The study offers a comparison of multiple state-of-the-art LLMs, analyzing susceptibility to issues such as hallucination or incomplete outputs. This evaluation identifies models best suited for constraint generation, with particular attention to the impact of model size and training scope on overall performance.

4. Parameters tuning insights and trade-offs

Through experiments with search and rerank parameters of the RAG pipeline, the research identifies diminishing returns beyond certain thresholds, providing recommendations for balancing accuracy with resource consumption.

5. Foundation for constraint generation using AI

Beyond immediate results, this dissertation establishes a basis for automatic constraint generation and a method for their evaluation, providing a framework that future research and industry efforts can build upon.

7.3 Future Work

This section highlights potential directions for advancing research and development that became apparent throughout the course of this work. Although the results underscore the value of combining RAG with prompt engineering for constraint generation, they also reveal certain gaps that merit further study.

This study focuses on a specific set of models, so it is advisable to conduct experiments with a broader scope of architectures. In this way, it will be possible to identify more precisely which model can present the best relationship between performance, inference time and suitability for constraint generation.

While RAG has already demonstrated clear advantages, there is room to examine additional parameters, such as chunk size and overlap. Investigating these factors could uncover configurations that further enhance accuracy, relevance, and processing efficiency, strengthening the robustness and practicality of the overall system.

Given that constraining designs is a specialized domain in ASIC digital design, further fine-tuning is suggested, employing a specific dataset (digital design and constraint set and tool's user guides). This approach, when paired with RAG, would ensure the models produce outputs grounded in accurate, up-to-date references while gaining a deeper understanding of the intricacies of constraint development.

Finally, future work could explore deeper integration with EDA tools themselves, enabling real-time feedback loops between the model and tools. Such an approach could allow the system to automatically adjust or regenerate constraints based on tool output (warnings, violations and errors), moving toward a closed-loop, intelligent assistant for ASIC engineers.

By pursuing these directions, this research can evolve from a proof of concept into a scalable, production-ready solution, helping bridge the gap between generative AI research and its application in the semiconductor industry.

References

- [1] J. Pan, G. Zhou, C.-C. Chang, I. Jacobson, J. Hu, and Y. Chen, “A survey of research in large language models for electronic design automation,” *arXiv*, Jan. 2025. DOI: 10.48550/arXiv.2501.09655. arXiv: 2501.09655 [cs.ET]. [Online]. Available: <https://arxiv.org/abs/2501.09655>.
- [2] M. Waller, “Standard design constraints: The next productivity boost for custom design,” *EE Times*, 2024, Accessed: 2025-07-27. [Online]. Available: <https://www.eetimes.com/standard-design-constraints-the-next-productivity-boost-for-custom-design/>.
- [3] Synopsys Newsroom, *Market Leaders Collaborate with Synopsys to Realize Gains of Generative AI Across Synopsys.ai Full EDA Stack*, Nov. 2023. [Online]. Available: <https://news.synopsys.com/2023-11-27-Market-Leaders-Collaborate-with-Synopsys-to-Realize-Gains-of-Generative-AI-Across-Synopsys-ai-Full-EDA-Stack>.
- [4] V. Taraate, *Digital Logic Design Using Verilog*. New Delhi: Springer India, 2016. DOI: 10.1007/978-81-322-2791-5.
- [5] Synopsys. “What is static timing analysis?” Accessed: 2025-07-27. (2025), [Online]. Available: <https://www.synopsys.com/glossary/what-is-static-timing-analysis.html>.
- [6] M. M. Rakshith and S. S. Hiremath, *Design and analysis of clock domain crossing using vc spyglass tool*, Accessed: 2025-07-27, 2025. [Online]. Available: https://d197for5662%20m48.cloudfront.net/documents/publicationstatus/174790/preprint_pdf/cd69ea62611e01e0034ec76ee52ba062.pdf.
- [7] M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*. New York, NY, USA: Springer Science+Business Media, 2012. DOI: 10.1007/978-1-4614-0397-5.
- [8] Philips Semiconductors, “A metastability primer,” Philips Semiconductors, Tech. Rep. AN219, Nov. 1989, Application Note. [Online]. Available: <https://www.nxp.com/docs/en/application-note/AN219.pdf>.
- [9] L. R. Marino, “General theory of metastable operation,” *IEEE Transactions on Computers*, vol. C-30, no. 2, pp. 107–115, 1981.
- [10] J. A. Baktash, M. Dawodi, and ChatGPT, “GPT-4: A review on advancements and opportunities in natural language processing,” *Journal of Electrical and Electronic Engineering*, vol. 2, no. 4, Dec. 2023. DOI: 10.33140/JEEE.02.04.19.
- [11] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, Sep. 2013. DOI: 10.48550/arXiv.1301.3781. [Online]. Available: <https://arxiv.org/abs/1301.3781>.

- [12] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need,” *arXiv preprint arXiv:1706.03762*, 2017. DOI: 10.48550/arXiv.1706.03762. [Online]. Available: <https://arxiv.org/abs/1706.03762>.
- [13] Z. Wang, Z. Chu, T. Viet Doan, S. Ni, M. Yang, and W. Zhang, “History, development, and principles of large language models—an introductory survey,” *arXiv preprint arXiv:2402.06853*, Feb. 2024. DOI: 10.48550/arXiv.2402.06853. [Online]. Available: <https://arxiv.org/abs/2402.06853>.
- [14] Y. Bengio, R. Ducharme, and P. Vincent, “A neural probabilistic language model,” *Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003.
- [15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *arXiv preprint arXiv:1810.04805*, 2019. DOI: 10.48550/arXiv.1810.04805. [Online]. Available: <https://arxiv.org/abs/1810.04805>.
- [16] J. Achiam, S. Adler, S. Agarwal, *et al.*, “GPT-4 Technical Report,” *arXiv preprint arXiv:2303.08774*, 2023. [Online]. Available: <https://arxiv.org/abs/2303.08774>.
- [17] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, *Language models are unsupervised multitask learners*, OpenAI Technical Report, 2019.
- [18] J. Kaplan, S. McCandlish, T. Henighan, *et al.*, “Scaling laws for neural language models,” *arXiv preprint arXiv:2001.08361*, Jan. 2020. DOI: 10.48550/arXiv.2001.08361. [Online]. Available: <https://arxiv.org/abs/2001.08361>.
- [19] J. Hoffmann, S. Borgeaud, A. Mensch, *et al.*, “Training compute-optimal large language models,” *arXiv preprint arXiv:2203.15556*, Mar. 2022. DOI: 10.48550/arXiv.2203.15556. [Online]. Available: <https://arxiv.org/abs/2203.15556>.
- [20] T. B. Brown, B. Mann, N. Ryder, *et al.*, “Language models are few-shot learners,” *arXiv preprint arXiv:2005.14165*, Jul. 2020. DOI: 10.48550/arXiv.2005.14165. [Online]. Available: <https://arxiv.org/abs/2005.14165>.
- [21] J. Wei, X. Wang, D. Schuurmans, *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 24 824–24 837.
- [22] P. Lewis, E. Perez, A. Piktus, *et al.*, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” *arXiv preprint arXiv:2005.11401*, Apr. 2021. DOI: 10.48550/arXiv.2005.11401. [Online]. Available: <https://arxiv.org/abs/2005.11401>.
- [23] Z. Zhang, A. Zhang, M. Li, and A. Smola, *Automatic chain of thought prompting in large language models*, <https://arxiv.org/abs/2308.09557>, Preprint, 2023.
- [24] Anthropic, *Claude 4*, <https://www.anthropic.com/news/claude-4>, Accessed: 2024-06-27, 2024.
- [25] Mistral AI, *Introducing Magistral*, <https://mistral.ai/news/magistral>, Accessed: 2024-06-27, 2024.
- [26] M. Abdin, S. Zhang, A. Celikyilmaz, *et al.*, “Phi-4-reasoning technical report,” *arXiv preprint arXiv:2504.21318*, Apr. 2025. DOI: 10.48550/arXiv.2504.21318. [Online]. Available: <https://arxiv.org/abs/2504.21318>.

- [27] L. Chen, M. Zaharia, and J. Zou, “FrugalGPT: How to use large language models while reducing cost and improving performance,” *arXiv preprint arXiv:2305.05176*, May 2023. DOI: 10.48550/arXiv.2305.05176. [Online]. Available: <https://arxiv.org/abs/2305.05176>.
- [28] OpenAI, *GPT-4.1*, <https://openai.com/index/gpt-4-1/>, Accessed: 2024-06-27, 2024.
- [29] Q. Team, *Qwen3 technical report*, 2025. arXiv: 2505.09388 [cs.CL]. [Online]. Available: <https://arxiv.org/abs/2505.09388>.
- [30] G. Team *et al.*, “Gemma 3 technical report,” arXiv, Tech. Rep. arXiv:2503.19786, Mar. 2025, Accessed: 2024-06-27. DOI: 10.48550/arXiv.2503.19786. [Online]. Available: <https://arxiv.org/abs/2503.19786>.
- [31] Meta AI, *Introducing llama 3.1: Our most capable models to date*, Meta AI Blog, Accessed via ai.meta.com/blog/meta-llama-3-1/, Jul. 2024.
- [32] Unsloth, *Magistral: How to run and fine-tune*, <https://docs.unsloth.ai/basics/tutorials-how-to-fine-tune-and-run-llms/magistral-how-to-run-and-fine-tune>, Accessed: 2024-06-27, 2024.
- [33] D. Hendrycks, C. Burns, S. Basart, *et al.*, “Measuring massive multitask language understanding,” *arXiv preprint arXiv:2009.03300*, 2020. [Online]. Available: <https://arxiv.org/abs/2009.03300>.
- [34] C. E. Jimenez, J. Yang, A. Wettig, *et al.*, “SWE-bench: Can language models resolve real-world github issues?” In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: <https://openreview.net/forum?id=VTF8yNQm66>.
- [35] Stanford & Anthropic Collaboration, *Terminal-Bench: A benchmark for AI agents in terminal environments*, <https://www.tbench.ai/>, 2025.
- [36] Vals AI, Inc., *Aime benchmark – vals ai*, <https://www.vals.ai/benchmarks/aime-2025-05-30>, Publicado em 30 de maio de 2025; avaliação de modelos em questões da AIME (2024–2025), May 2025.
- [37] M. Chen, J. Tworek, H. Jun, *et al.*, “Evaluating large language models trained on code,” 2021. arXiv: 2107.03374 [cs.LG].
- [38] J. Blocklove, S. Garg, R. Karri, and H. Pearce, “Chip-chat: Challenges and opportunities in conversational hardware design,” in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, Sep. 2023, pp. 1–6. DOI: 10.1109/MLCAD58807.2023.10299874.
- [39] K. Chang *et al.*, “ChipGPT: How far are we from natural language hardware design,” *arXiv preprint arXiv:2305.14019*, Jun. 2023. DOI: 10.48550/arXiv.2305.14019. [Online]. Available: <https://arxiv.org/abs/2305.14019>.
- [40] X. Wang *et al.*, “ChatCPU: An agile CPU design and verification platform with llm,” in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, San Francisco, CA, USA: ACM, Jun. 2024, pp. 1–6. DOI: 10.1145/3649329.3658493.
- [41] Y.-D. Tsai, M. Liu, and H. Ren, “RTLFixer: Automatically fixing rtl syntax errors with large language models,” *arXiv preprint arXiv:2311.16543*, May 2024. DOI: 10.48550/arXiv.2311.16543. [Online]. Available: <https://arxiv.org/abs/2311.16543>.

- [42] K. Thorat *et al.*, “Advanced large language model (LLM)-driven verilog development: Enhancing power, performance, and area optimization in code synthesis,” *arXiv*, Jan. 2024. DOI: 10.48550/arXiv.2312.01022. arXiv: 2312.01022 [cs.AR]. [Online]. Available: <https://arxiv.org/abs/2312.01022>.
- [43] U. Kamath, K. Keenan, G. Somers, and S. Sorenson, “Retrieval-augmented generation,” in *Large Language Models: A Deep Dive*, Cham: Springer Nature Switzerland, 2024, pp. 275–313. DOI: 10.1007/978-3-031-65647-7_7.
- [44] N. Reimers and I. Gurevych, “Sentence-BERT: Sentence embeddings using siamese bert-networks,” *arXiv*, Aug. 2019. DOI: 10.48550/arXiv.1908.10084. arXiv: 1908.10084 [cs.CL]. [Online]. Available: <https://arxiv.org/abs/1908.10084>.
- [45] AutoRAG Contributors, *Autorag documentation*, <https://docs.auto-rag.com/index.html>, Accessed: 2025-05-29, 2025.
- [46] Y. Gao, Y. Xiong, X. Gao, *et al.*, “Retrieval-augmented generation for large language models: A survey,” *arXiv preprint arXiv:2312.10997*, Mar. 2024. DOI: 10.48550/arXiv.2312.10997. [Online]. Available: <https://arxiv.org/abs/2312.10997>.
- [47] K. Shuster, S. Poff, M. Chen, D. Kiela, and J. Weston, “Retrieval augmentation reduces hallucination in conversation,” in *Findings of the Association for Computational Linguistics: EMNLP 2021*, Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021, pp. 3784–3803. DOI: 10.18653/v1/2021.findings-emnlp.320.
- [48] A. Lozano, S. L. Fleming, C. Chiang, and N. Shah, “Clinfo.ai: An open-source retrieval-augmented large language model system for answering medical questions using scientific literature,” *arXiv preprint*, 2023, arXiv:2310.16146. [Online]. Available: <http://arxiv.org/abs/2310.16146> (visited on 10/26/2024).
- [49] X. Li, Z. Li, C. Shi, *et al.*, *Alphafin: Benchmarking financial analysis with retrieval-augmented stock-chain framework*, 2024. arXiv: 2403.12582 [cs.CL]. [Online]. Available: <https://arxiv.org/abs/2403.12582>.
- [50] V. A. Lovtsov and M. A. Skvortsova, “Automated mobile operator customer service using large language models combined with RAG system,” in *2025 7th International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE)*, 2025, pp. 1–6. DOI: 10.1109/REEPE63962.2025.10971107.
- [51] Google Labs, *Notebooklm: Ai research tool & thinking partner*, <https://notebooklm.google/>, 2025.
- [52] R. Tozuka, H. Onishi, M. Anzai, Y. Kurata, S. Fujimoto, and K. Yasaka, “Application of NotebookLM, a Large Language Model with Retrieval-Augmented Generation, for Lung Cancer Staging,” *Japanese Journal of Radiology*, Nov. 2024. DOI: 10.1007/s11604-024-01705-1. [Online]. Available: <https://doi.org/10.1007/s11604-024-01705-1>.
- [53] N. Muennighoff, N. Tazi, L. Magne, and N. Reimers, “MTEB: Massive text embedding benchmark,” *arXiv preprint arXiv:2210.07316*, 2022. DOI: 10.48550/ARXIV.2210.07316. [Online]. Available: <https://arxiv.org/abs/2210.07316>.
- [54] W. Shi, S. Min, M. Yasunaga, *et al.*, *Replug: Retrieval-augmented black-box language models*, 2023. arXiv: 2301.12652 [cs.CL].
- [55] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi, *Self-RAG: Learning to retrieve, generate, and critique through self-reflection*, 2023. arXiv: 2310.11511 [cs.CL].

- [56] Synopsys. “What is clock domain crossing?” Accessed: 2025-07-27. (2025), [Online]. Available: <https://www.synopsys.com/blogs/chip-design/what-is-clock-domain-crossing.html>.
- [57] Synopsys. “Design compiler ultra.” Accessed: 2025-07-27. (2025), [Online]. Available: <https://www.synopsys.com/implementation-and-signoff/rtl-synthesis%20is-test/dc-ultra.html>.
- [58] LangChain, *Langchain official website*, <https://www.langchain.com/>, Accessed: 2025-07-27.
- [59] Deepset, *Haystack official website*, <https://haystack.deepset.ai/>, Accessed: 2025-07-27, 2024.
- [60] LlamaIndex, *Llamaindex official website*, <https://www.llamaindex.ai/>, Accessed: 2025-07-27, 2024.
- [61] Elastic, *Elasticsearch*, <https://github.com/elastic/elasticsearch>, Accessed: 2025-07-27, 2025.
- [62] Facebook Research, *Faiss: A library for efficient similarity search and clustering of dense vectors*, <https://github.com/facebookresearch/faiss>, Accessed: 2025-07-27, 2025.
- [63] Vellum, *Llamaindex vs. langchain: Comparison for production-ready RAG applications*, Accessed: 2025-07-27, 2024.
- [64] S. Lee, A. Shakir, D. Koenig, and J. Lipp. “Open source strikes bread - new fluffy embeddings model.” (2024), [Online]. Available: <https://www.mixedbread.ai/blog/mxbai-embed-large-v1>.
- [65] Nomic AI, *Nomic-embed-text-v1*, <https://huggingface.co/nomic-ai/nomic-embed-text-v1>, Accessed: 2025-07-27, 2025.
- [66] Z. Nussbaum, J. X. Morris, B. Duderstadt, and A. Mulyar, *Nomic embed: Training a reproducible long context text embedder*, 2024. arXiv: 2402.01613 [cs.CL].
- [67] Unsloth, *Llama-3.1-8b-instruct-gguf*, <https://huggingface.co/unsloth/Llama-3.1-8B-Instruct-GGUF>, Accessed: 2025-07-27, 2025.
- [68] Unsloth, *Gemma-3-27b-it*, <https://huggingface.co/unsloth/gemma-3-27b-it>, Accessed: 2025-07-27, 2025.
- [69] Unsloth, *Mistral-small-3.1-24b-instruct-2503*, <https://huggingface.co/unsloth/Mistral-Small-3.1-24B-Instruct-2503>, Accessed: 2025-07-27, 2025.
- [70] Unsloth, *Llama-3.3-70b-instruct-gguf*, <https://huggingface.co/unsloth/Llama-3.3-70B-Instruct-GGUF>, Accessed: 2025-07-27, 2025.
- [71] Unsloth, *Deepseek-r1-distill-llama-70b-gguf*, <https://huggingface.co/unsloth/DeepSeek-R1-Distill-Llama-70B-GGUF>, Accessed: 2025-07-27, 2025.
- [72] QwenLM Team. “Qwen2.5 coder family.” Accessed: 2025-07-27. (2025), [Online]. Available: <https://qwenlm.github.io/blog/qwen2.5-coder-family/>.
- [73] J. Risch, T. Möller, J. Gutsch, and M. Pietsch, “Semantic answer similarity for evaluating question answering models,” *arXiv preprint arXiv:2108.06130*, 2021. DOI: 10.48550/arXiv.2108.06130. arXiv: 2108.06130 [cs.CL]. [Online]. Available: <https://arxiv.org/abs/2108.06130>.

- [74] LlamaIndex, *Correctness evaluation — llamaindex documentation*, https://docs.llamaindex.ai/en/stable/examples/evaluation/correctness_eval/, Accessed: 2025-07-27, 2024.
- [75] A. K. et al., “Better zero-shot reasoning with role-play prompting,” 2024. arXiv: 2308.07702 [cs.CL].
- [76] Synopsys® *timing constraints and optimization user guide*, version U-2002.12, Accessed: 2025-07-27, Synopsys, Inc.

Appendix A

Benchmark Designs

A.1 Reset synchronizer module

The `areset_sync` module is a reset synchronizer that ensures safe handling of asynchronous reset signals across all benchmarks designs.

Asynchronous resets, when de-asserted without alignment to the system clock, may cause metastability and unpredictable circuit behavior. This module mitigates that risk by passing the reset through a configurable chain of flip-flops (`STAGES`, typically 2), thereby synchronizing the release of the reset to the target clock. The polarity of the reset is also parametrizable (`RST_POL`), supporting both active-low and active-high conventions. In practice, the reset is asserted immediately, while its de-assertion is delayed and aligned with the clock, ensuring a glitch-free and reliable reset signal for downstream logic.

```
1 module areset_sync #(
2     parameter STAGES = 2 ,
3     parameter RST_POL = 1'b0
4 )
5 (
6     input  logic clk ,
7     input  logic i_rst_async ,
8     output logic o_rst_sync
9 ) ;
10
11 logic reset ;
12 assign reset = i_rst_async ^ RST_POL ;
13
14 (* ASYNC_REG = "TRUE" *)
15 logic [STAGES-1:0] sync_ff ;
16
17 always @(posedge clk or negedge reset) begin
18
19     if (!reset) begin
20         sync_ff <= {STAGES{RST_POL}} ;
21     end
22 end
```

```

22     else begin
23         sync_ff <= {sync_ff [STAGES-2 : 0], ~RST_POL} ;
24     end
25
26 end
27 assign o_rst_sync = sync_ff [STAGES-1] ;
28 endmodule

```

Listing A.1: Reset synchronizer module

A.2 Asynchronous FIFO Design

An asynchronous FIFO is a memory buffer that allows data to be written in one clock domain and read from another, making it a widely used structure in systems where components operate at different clock frequencies. The design ensures reliable data transfer across clock domains while avoiding metastability and data corruption.

A.2.1 Architecture Overview

The design implemented is composed by the following modules:

- **a_fifo:** The top-level wrapper that instantiates all submodules and manages the overall connectivity between write and read sides (Listing A.2).
- **fifo_mem:** The memory array used to store the FIFO data. The write pointer selects the write address while the read pointer selects the read address (Listing A.3).
- **rpitr_handler:** Generates and updates the binary and Gray-coded read pointers. It also computes the `empty` flag (Listing A.4).
- **wptr_handler:** Generates and updates the binary and Gray-coded write pointers. It also computes the `full` flag (Listing A.5).
- **synchronizer:** A two-stage flip-flop synchronizer used to safely transfer Gray-coded pointers across clock domains (Listing A.6).
- **areset_sync:** A reset synchronizer (Listing A.1).

A.2.2 High-Level Operation

On the *write side*, data is stored into the memory when `w_en` is asserted and the FIFO is not full. The write pointer is incremented accordingly, with both its binary and Gray-coded versions being maintained. On the *read side*, data is retrieved when `r_en` is asserted and the FIFO is not empty. Similarly, the read pointer is updated in both binary and Gray-coded form.

To safely exchange pointer information between the two clock domains, the Gray-coded pointers are synchronized using the `synchronizer` module. The `full` and `empty` flags are then

generated by comparing the synchronized pointers, ensuring that read and write operations never overlap incorrectly.

A.2.3 Gray Code in FIFOs

A key feature of the implemented asynchronous FIFO is the use of **Gray coding** for pointer representation. In Gray code, only one bit changes between successive values, which minimizes the risk of sampling errors when pointers cross between different clock domains. If binary counters were used directly, multiple bit transitions could occur simultaneously, leading to metastability or incorrect comparisons (Table A.1). By using Gray code and synchronizers, the FIFO guarantees reliable detection of `full` and `empty` conditions even under asynchronous clocking.

Table A.1: 3-bit mapping: Binary and Gray codes

Decimal	Binary	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

A.2.4 Verilog Implementation

The Verilog source code of the asynchronous FIFO design and its submodules is included below.

```

1 module a_fifo #(parameter DEPTH=8, DATA_WIDTH=8) (
2     input  wclk, wrst_n,
3     input  rclk, rrst_n,
4     input  w_en, r_en,
5     input  [DATA_WIDTH-1:0] data_in,
6     output [DATA_WIDTH-1:0] data_out,
7     output full, empty
8 );
9
10 parameter PTR_WIDTH = $clog2(DEPTH);
11
12 wire [PTR_WIDTH:0] g_wptr_sync, g_rptr_sync;
13 wire [PTR_WIDTH:0] b_wptr, b_rptr;
14 wire [PTR_WIDTH:0] g_wptr, g_rptr;
15 wire w_en_sync, r_en_sync;
16

```

```

17  wire [PTR_WIDTH-1:0] waddr, raddr;
18  wire wrst_n_sync, rrst_n_sync;
19
20  areset_sync wclk_rst_sync(
21    .clk(wclk),
22    .i_rst_async(wrst_n),
23    .o_rst_sync(wrst_n_sync)
24  );
25
26  areset_sync rclk_rst_sync(
27    .clk(rclk),
28    .i_rst_async(rrst_n),
29    .o_rst_sync(rrst_n_sync)
30  );
31
32  wptr_handler #(.PTR_WIDTH(PTR_WIDTH)) wptr_h(
33    .wclk(wclk),
34    .wrst_n(wrst_n_sync),
35    .w_en(w_en),
36    .g_rptr_sync(g_rptr_sync),
37    .b_wptr(b_wptr),
38    .g_wptr(g_wptr),
39    .full(full)
40  );
41
42  rpтр_handler #(.PTR_WIDTH(PTR_WIDTH)) rpтр_h(
43    .rclk(rclk),
44    .rrst_n(rrst_n_sync),
45    .r_en(r_en),
46    .g_wptr_sync(g_wptr_sync),
47    .b_rptr(b_rptr),
48    .g_rptr(g_rptr),
49    .empty(empty)
50  );
51
52  synchronizer #(.WIDTH(PTR_WIDTH)) sync_rptr(
53    .clk(rclk),
54    .rst(rrst_n_sync),
55    .d1(g_wptr),
56    .d0(g_wptr_sync)
57  );
58
59  synchronizer #(.WIDTH(PTR_WIDTH)) sync_wptr(
60    .clk(wclk),
61    .rst(wrst_n_sync),
62    .d1(g_rptr),
63    .d0(g_rptr_sync)
64  );
65

```

```

66     fifo_mem fifom(
67         .wclk(wclk),
68         .w_en(w_en),
69         .wrst_n(wrst_n_sync),
70         .b_wptr(b_wptr),
71         .b_rptr(b_rptr),
72         .data_in(data_in),
73         .full(full),
74         .data_out(data_out)
75     );
76
77 endmodule

```

Listing A.2: Implementation of a_fifo module.

```

1 module fifo_mem #(parameter DEPTH=8, DATA_WIDTH=8, PTR_WIDTH=3) (
2     input wclk, w_en, wrst_n,
3     input [PTR_WIDTH:0] b_wptr, b_rptr,
4     input [DATA_WIDTH-1:0] data_in,
5     input full,
6     output reg [DATA_WIDTH-1:0] data_out
7 );
8     reg [DATA_WIDTH-1:0] fifo[0:DEPTH-1];
9
10    always@(posedge wclk or negedge wrst_n) begin
11        if(!wrst_n) begin
12            fifo <= '{default: {DATA_WIDTH{1'b0}}};
13        end else begin
14            if(w_en & !full) begin
15                fifo[b_wptr[PTR_WIDTH-1:0]] <= data_in;
16            end
17        end
18    end
19
20    assign data_out = fifo[b_rptr[PTR_WIDTH-1:0]];
21
22 endmodule

```

Listing A.3: Implementation of fifo_mem module.

```

1
2 module rp_ptr_handler #(parameter PTR_WIDTH) (
3     input rclk, rrst_n, r_en,
4     input [PTR_WIDTH:0] g_wptr_sync,
5     output reg [PTR_WIDTH:0] b_rptr, g_rptr,
6     output reg empty
7 );
8
9     reg [PTR_WIDTH:0] b_rptr_next;
10    reg [PTR_WIDTH:0] g_rptr_next;

```

```

11  wire empty;
12
13  assign b_rptr_next = b_rptr+(r_en & !empty) & 4'hF;
14  assign g_rptr_next = (b_rptr_next >>1)^b_rptr_next;
15  assign empty = (g_wptr_sync == g_rptr_next);
16
17  always@(posedge rclk or negedge rrst_n) begin
18      if(!rrst_n) begin
19          b_rptr <= {(PTR_WIDTH+1){1'b0}};
20          g_rptr <= {(PTR_WIDTH+1){1'b0}};
21      end
22      else begin
23          b_rptr <= b_rptr_next;
24          g_rptr <= g_rptr_next;
25      end
26  end
27
28  always@(posedge rclk or negedge rrst_n) begin
29      if(!rrst_n) empty <= 1'b1;
30      else          empty <= empty;
31  end
32 endmodule

```

Listing A.4: Implementation of rpтр_handler module.

```

1  module wptr_handler #(parameter PTR_WIDTH) (
2      input wclk,  wrst_n, w_en,
3      input [PTR_WIDTH:0] g_rptr_sync,
4      output reg [PTR_WIDTH:0] b_wptr, g_wptr,
5      output reg full
6  );
7
8      reg [PTR_WIDTH:0] b_wptr_next;
9      reg [PTR_WIDTH:0] g_wptr_next;
10
11     reg wrap_around;
12     wire wfull;
13
14     assign b_wptr_next = b_wptr+(w_en & !full) & 4'hF;
15     assign g_wptr_next = (b_wptr_next >>1)^b_wptr_next;
16
17     always@(posedge wclk or negedge wrst_n) begin
18         if(!wrst_n) begin
19             b_wptr <= {(PTR_WIDTH+1){1'b0}};
20             g_wptr <= {(PTR_WIDTH+1){1'b0}};
21         end
22         else begin
23             b_wptr <= b_wptr_next;
24             g_wptr <= g_wptr_next;
25         end

```

```

26 end
27
28 always@(posedge wclk or negedge wrst_n) begin
29     if(!wrst_n) full <= 1'b0;
30     else          full <= wfull;
31 end
32
33 assign wfull = (g_wptr_next == {~g_rptr_sync[PTR_WIDTH:PTR_WIDTH-1],
34     g_rptr_sync[PTR_WIDTH-2:0]});
35
36 endmodule

```

Listing A.5: Implementation of wptr_handler module.

```

1 module synchronizer #(parameter WIDTH=3) (input clk, rst, [WIDTH:0] d1, output reg
2 [WIDTH:0] d0);
3     reg [WIDTH:0] q1;
4
5     always@(posedge clk or negedge rst) begin
6         if(!rst) begin
7             q1 <= {(WIDTH+1){1'b0}};
8             d0 <= {(WIDTH+1){1'b0}};
9         end
10        else begin
11            q1 <= d1;
12            d0 <= q1;
13        end
14    end
15 endmodule

```

Listing A.6: Implementation of synchronizer module.

A.3 Synchronous FIFO Design

Unlike the asynchronous FIFO, which operates with independent read and write clocks, the synchronous FIFO is entirely managed within a single clock domain. This makes its architecture simpler, while still maintaining the same fundamental FIFO behavior.

Architecture Overview

The synchronous FIFO design is composed of the following modules:

- **fifo:** The top-level wrapper that connects the reset synchronizer and the FIFO controller, exposing the external interface for data and control signals (Listing A.7).
- **fifo_ctrl:** The core of the design, implementing the FIFO memory array, read/write pointer updates, and the generation of the `full` and `empty` flags (Listing A.8).
- **areset_sync:** A reset synchronizer (Listing A.1).

A.3.1 High-Level Operation

The FIFO memory is organized as an array of depth `DEPTH` and width `DATA_WIDTH`. A binary write pointer (`w_ptr`) selects the location for incoming data, while a binary read pointer (`r_ptr`) selects the element to be output. On each clock cycle:

- If `w_en` is asserted and the FIFO is not full, the input data is written at the location indicated by `w_ptr`, which is then incremented.
- If `r_en` is asserted and the FIFO is not empty, the element at `r_ptr` is output and the pointer is incremented.

The `full` and `empty` flags are generated by comparing the current positions of the read and write pointers, preventing illegal operations.

A.3.2 Verilog Implementation

The Verilog source code of the synchronous FIFO design and its submodules is included below.

```

1 module fifo #(parameter DEPTH=8, DATA_WIDTH=8) (
2     input clk,
3     input rst_n,
4     input w_en,
5     input r_en,
6     input [DATA_WIDTH-1:0] data_in,
7     output [DATA_WIDTH-1:0] data_out,
8     output full,
9     output empty
10 );
11     wire rst_n_sync;
12
13     areset_sync rst_synchronizer (
14         .clk(clk),
15         .i_rst_async(rst_n),
16         .o_rst_sync(rst_n_sync)
17     );
18
19     fifo_ctrl #(
20         .DEPTH(DEPTH),
21         .DATA_WIDTH(DATA_WIDTH)
22     ) fifo_ctrl (
23         .clk(clk),
24         .rst_n(rst_n_sync),
25         .w_en(w_en),
26         .r_en(r_en),
27         .data_in(data_in),
28         .data_out(data_out),
29         .full(full),
30         .empty(empty)

```

```

31 );
32
33 endmodule

```

Listing A.7: Implementation of fifo module.

```

1 module fifo_ctrl #(parameter DEPTH=8, DATA_WIDTH=8) (
2     input  clk,
3     input  rst_n,
4     input  w_en,
5     input  r_en,
6     input  [DATA_WIDTH-1:0] data_in,
7     output reg [DATA_WIDTH-1:0] data_out,
8     output full,
9     output empty
10 );
11
12     reg [$clog2(DEPTH)-1:0] w_ptr, r_ptr;
13     reg [DATA_WIDTH-1:0] fifo[DEPTH];
14
15     always @(posedge clk or negedge rst_n) begin
16         if (!rst_n) begin
17             w_ptr <= 0;
18             r_ptr <= 0;
19             data_out <= 0;
20             fifo <= '{default: {DATA_WIDTH{1'b0}}};
21         end else begin
22             if (w_en & !full) begin
23                 fifo[w_ptr] <= data_in;
24                 w_ptr <= w_ptr + 1;
25             end
26             if (r_en & !empty) begin
27                 data_out <= fifo[r_ptr];
28                 r_ptr <= r_ptr + 1;
29             end
30         end
31     end
32
33     assign full = ((w_ptr+1'b1) == r_ptr);
34     assign empty = (w_ptr == r_ptr);
35
36 endmodule

```

Listing A.8: Implementation of fifo_ctrl module.

A.4 Dual-Port RAM Design

A dual-port RAM allows two independent ports to perform read and write operations simultaneously, enabling parallel access to memory.

A.4.1 Architecture Overview

The design is organized into the following modules:

- **dualport_ram:** The top-level wrapper that instantiates both memory ports (`ram_port_a` and `ram_port_b`) and connects them to independent clocks and reset signals (Listing A.9).
- **ram_port_a:** Implements the memory operations for Port A (Listing A.10).
- **ram_port_b:** Implements the memory operations for Port B (Listing A.11).
- **areset_sync:** A reset synchronizer (Listing A.1).

A.4.2 High-Level Operation

The memory array for each port consists of eight 128-bit words, indexed by a 3-bit address. Each port has its own:

- clock signal (`clk_a` or `clk_b`),
- write enable (`wr_en_a`, `wr_en_b`),
- 3-bit address (`addr_a`, `addr_b`),
- 128-bit input (`data_in_a`, `data_in_b`) and output (`data_out_a`, `data_out_b`) data buses.

On each clock cycle:

- If the write enable is asserted, the input data is written into the memory location specified by the address.
- Otherwise, the data stored at the given address is read and presented on the output.

Both ports can operate simultaneously and independently, provided they do not attempt conflicting writes to the same address at the same time. Resetting either port clears its memory and output registers.

A.4.3 Verilog Implementation

The Verilog source code of the dual-port RAM design and its submodules is included below.

```

1 module dualport_ram(
2     input          clk_a,
3     input          clk_b,
4     input          rst,
5     input          wr_en_a,
6     input          wr_en_b,
7     input [2:0]    addr_a,
8     input [2:0]    addr_b,
```

```

9     input      [127:0] data_in_a,
10    input      [127:0] data_in_b,
11    output     [127:0] data_out_a,
12    output     [127:0] data_out_b
13 );
14
15 wire rst_a_sync;
16 wire rst_b_sync;
17
18 areset_sync resets_sync_a(
19     .clk(clk_a),
20     .i_rst_async(rst),
21     .o_rst_sync(rst_a_sync)
22 );
23
24 areset_sync resets_sync_b(
25     .clk(clk_b),
26     .i_rst_async(rst),
27     .o_rst_sync(rst_b_sync)
28 );
29
30 ram_port_a port_a(
31     .clk_a(clk_a),
32     .rst_a_n(rst_a_sync),
33     .wr_en_a(wr_en_a),
34     .addr_a(addr_a),
35     .data_in_a(data_in_a),
36     .data_out_a(data_out_a)
37 );
38
39 ram_port_b port_b(
40     .clk_b(clk_b),
41     .rst_b_n(rst_b_sync),
42     .wr_en_b(wr_en_b),
43     .addr_b(addr_b),
44     .data_in_b(data_in_b),
45     .data_out_b(data_out_b)
46 );
47
48 endmodule

```

Listing A.9: Implementation of dualport_ram module.

```

1 module ram_port_a (
2     input clk_a,
3     input rst_a_n,
4     input wr_en_a,
5     input [2:0] addr_a,
6     input [127:0] data_in_a,
7     output reg [127:0] data_out_a

```

```

8 );
9   reg [127:0] ram_vector_a [7:0];
10  integer i;
11
12  always @(posedge clk_a or negedge rst_a_n) begin
13      if (!rst_a_n) begin
14          data_out_a <= 128'b0;
15          for (i = 0; i < 8; i = i + 1)
16              ram_vector_a[i] <= 128'b0;
17      end else begin
18          if (wr_en_a)
19              ram_vector_a[addr_a] <= data_in_a;
20          else
21              data_out_a <= ram_vector_a[addr_a];
22      end
23  end
24
25  endmodule

```

Listing A.10: Implementation of ram_port_a module.

```

1  module ram_port_b (
2      input clk_b,
3      input rst_b_n,
4      input wr_en_b,
5      input [2:0] addr_b,
6      input [127:0] data_in_b,
7      output reg [127:0] data_out_b
8  );
9      reg [127:0] ram_vector_b [7:0];
10     integer i;
11
12     always @(posedge clk_b or negedge rst_b_n) begin
13         if (!rst_b_n) begin
14             data_out_b <= 128'b0;
15             for (i = 0; i < 8; i = i + 1)
16                 ram_vector_b[i] <= 128'b0;
17         end else begin
18             if (wr_en_b)
19                 ram_vector_b[addr_b] <= data_in_b;
20             else
21                 data_out_b <= ram_vector_b[addr_b];
22         end
23     end
24 end
25
26 endmodule

```

Listing A.11: Implementation of ram_port_b module.

A.5 Single-Port RAM Design

The single-port RAM is a compact memory block with one shared address and data port. Operating in a single clock domain, it supports both write and read cycles on the same port and delivers the read data with a one-cycle latency (synchronous read).

A.5.1 Architecture Overview

The design is organized into the following modules:

- **singleport_ram:** Top-level wrapper that synchronizes the asynchronous reset and instantiates the core memory port (Listing A.12).
- **port_ram:** Core single-port memory implementation. Internally, it instantiates an array (depth= 8 and width= 128), uses a registered read address pipeline for synchronous readout, and supports writes via `wr_en` (Listing A.3).
- **areset_sync:** A reset synchronizer (Listing A.1).

A.5.2 High-Level Operation

The RAM exposes a single port with signals: clock (`clk`), reset (`rst`), write enable (`wr_en`), a 3-bit address (`addr`), a 128-bit input (`data_in`), and a 128-bit output (`data_out`). The memory array is modeled as `ram_vector[7:0]` (eight 128-bit words). The design performs:

- **Write:** On a rising clock edge, if `wr_en` is asserted, the word at `ram_vector[addr]` is updated with `data_in`.
- **Read:** The read address is pipelined through two registers (`read_addr_aux` then `read_addr`). The output `data_out` is driven from `ram_vector[read_addr]`, resulting in a one-cycle read latency with respect to the applied `addr`.

A.5.3 Verilog Implementation

The Verilog source code of the single-port RAM and its submodules is included below.

```

1 module singleport_ram(
2     input  clk          ,
3     input  rst          ,
4     input  wr_en        ,
5     input  [2:0] addr   ,
6     input  [127:0] data_in ,
7     output [127:0] data_out
8 )
9     ;
10 reg [127:0] ram_vector [7:0] ;
11 reg [2:0] read_addr          ;

```

```

12 reg [2:0] read_addr_aux      ;
13 wire rst_sync                ;
14 integer i                    ;
15
16 areset_sync resets_sync(
17     .clk(clk),
18     .i_rst_async(rst),
19     .o_rst_sync(rst_sync)
20 );
21
22 port_ram port_inst(
23     .clk(clk),
24     .rst(rst_sync),
25     .wr_en(wr_en),
26     .addr(addr),
27     .data_in(data_in),
28     .data_out(data_out)
29 );
30
31 endmodule

```

Listing A.12: Implementation of ram_port_b module.

Core Port (port_ram.v)

```

1 module port_ram(
2     input clk,
3     input rst,
4     input wr_en,
5     input [2:0] addr,
6     input [127:0] data_in,
7     output reg [127:0] data_out
8 );
9
10 reg [127:0] ram_vector [7:0] ;
11 reg [2:0] read_addr          ;
12 reg [2:0] read_addr_aux      ;
13 wire rst_sync                ;
14 integer i                    ;
15
16 always @(posedge clk or negedge rst) begin
17     if (!rst) begin
18         data_out <= 128'b0;
19         for (i = 0; i < 8; i = i + 1) begin
20             ram_vector[i] <= 128'b0;
21         end
22         read_addr <= 3'b0;
23         read_addr_aux <= 3'b0;
24     end else begin

```

```

25     data_out <= ram_vector[read_addr];
26     if (wr_en) begin
27         ram_vector[addr] <= data_in;
28     end
29     read_addr_aux <= addr;
30     read_addr <= read_addr_aux;
31 end
32 end
33
34 endmodule

```

Listing A.13: Implementation of ram_port_b module.

A.6 Clock Divider Design

This clock divider is a small benchmark block used to derive a lower-frequency clock from a faster input clock. It operates in a single clock domain and divides the input frequency by a programmable integer factor `DIV_VALUE`.

A.6.1 Architecture Overview

The design is organized into the following modules:

- **clk_div:** Top-level wrapper that synchronizes the asynchronous reset and instantiates the counter module. It generates `clk_out` by thresholding the counter value (Listing A.14).
- **counter:** Parametrizable N up-counter (with $N = \text{DIV_VALUE}$) that counts from 0 to $N - 1$ (Listing A.15).
- **areset_sync:** A reset synchronizer (Listing A.1).

A.6.2 High-Level Operation

The parameter `DIV_VALUE` sets the division ratio N . The counter increments on each rising edge of `clk_in` and wraps to zero when it reaches $N - 1$. The output clock `clk_out` is derived by comparing the current count against $N/2$:

$$\text{clk_out} = \begin{cases} 0, & \text{if counter} < \left\lfloor \frac{N}{2} \right\rfloor \\ 1, & \text{otherwise} \end{cases}$$

This produces a period of exactly N input cycles, yielding an output frequency $f_{\text{out}} = f_{\text{in}}/N$. For even N , the duty cycle is 50%; for odd N , the duty cycle is slightly asymmetric (but still divides the frequency by N overall).

A.6.3 Verilog Implementation

The Verilog source code of the clock divider and its submodules is included below.

```

1 module clk_div #(
2     parameter DIV_VALUE = 2
3 ) (
4     input clk_in,
5     input reset,
6     output clk_out
7 );
8
9     wire [$clog2(DIV_VALUE)-1:0] counter;
10
11     wire reset_sync;
12
13     areset_sync rst_sync (
14         .clk(clk_in),
15         .i_rst_async(reset),
16         .o_rst_sync(reset_sync)
17     );
18
19     counter #(
20         .DIV_VALUE(DIV_VALUE)
21     ) counter_inst (
22         .clk_in(clk_in),
23         .reset(reset_sync),
24         .counter(counter)
25     );
26
27     assign clk_out = (counter < (DIV_VALUE/2)) ? 1'b0 : 1'b1;
28
29 endmodule

```

Listing A.14: Implementation of clk_div module.

```

1 module counter #(parameter DIV_VALUE = 2) (
2     input clk_in,
3     input reset,
4     output reg [$clog2(DIV_VALUE)-1:0] counter
5 );
6
7 always @(posedge clk_in or negedge reset) begin
8     if (!reset)
9         counter <= 0;
10    else if (counter == DIV_VALUE - 1)
11        counter <= 0;
12    else
13        counter <= counter + 1;
14 end
15

```

```
16 endmodule
```

Listing A.15: Implementation of counter module.

Appendix B

Design constraints

This chapter explores Synopsys Design Constraints that were exercised across the benchmark designs used in this work. The aim is not to be exhaustive, but to provide concise, reproducible snippets of design constraints.

B.1 Clock constraints

Two clocks are synchronous with respect to each other if they share a common source and have a fixed phase relationship. Clocks are defined by the `create_clock` command. For example consider the following `create_clock` commands:

```
1 prompt > create_clock -period 4 -name CK1 -waveform {0 2}
2 prompt > create_clock -period 4 -name CK2 -waveform {1 3}
3 prompt > create_clock -period 6 -name CK3 -waveform {2 3}
```

The created clocks specified in the commands have the following synchronized waveforms:

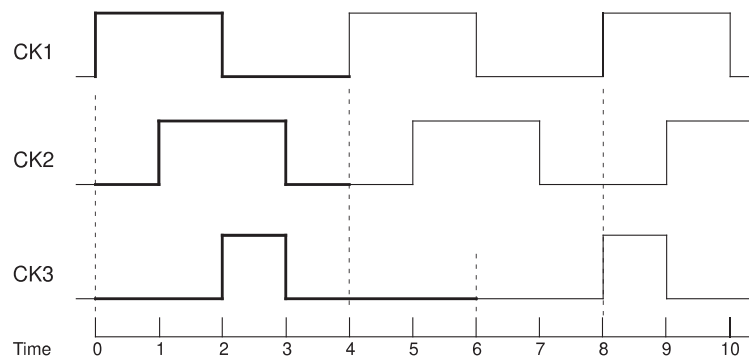


Figure B.1: Synchronous clock waveforms (from [76]).

In opposition, two clocks are asynchronous if they do not communicate with each other in the design. For example, a free-running, on-chip oscillator is asynchronous with respect to a system clock signal coming into the chip from the outside. Clock edges in the two clock domains can

occurs at any time with respect to each other. The following command is used to declare the asynchronous relationship between clocks:

```
1 prompt > set_clock_groups -asynchronous
```

In addition to the asynchronous relationship, clocks can be exclusive if they do not interact with each other. For example, a circuit might multiplex two different clock signals onto a clock line, one a fast clock for normal operation and the other a slow clock for low-power operation. Only one of the two clocks is enabled at any given time, so there is no interaction between the two clocks.

To prevent the tool from spending time analyzing the interaction between exclusive clocks, the following command should be used:

```
1 prompt > set_clock_groups -logically_exclusive
```

For example to declare CK1, CK2 and CK3 to be logically exclusive:

```
1 prompt > set_clock_groups -logically_exclusive -group {CK1} -group {CK2} -group {CK3}
}
```

This option causes the tool to suppress any logical checking between the clocks. However, it still computes crosstalk delta delays across coupling capacitor between the clocks, which is pessimistic because the clocks are never simultaneously present on the nets. To eliminate this pessimism, the clocks can be defined as physically exclusive.

```
1 prompt > set_clock_groups -physically_exclusive -group {CK1} -group {CK2} -group {CK3}
```

Moreover, a design might include clock dividers or other structures that produce a new clock from a master source clock. A clock that is generated by on-chip logic from another clock is called a generated clock.

In the case of a divide-by-2 generated clock the command `create_generated_clock` should be used. This command specifies the characteristics of an internally generated clock in terms of the master clock. The generated clock characteristics are based on the master clock. If the period or latency of the master clock changes, the generated clock also changes accordingly.

```
1 prompt > create_generated_clock -name DIVIDE -source [get_ports SYCLK] -divide_by 2
[get_pins FF1/Q]
```

B.2 Timing constraints

The `set_input_delay` command specifies the timing of external paths leading to an input port. The input delay is the arrival time of a transition at an input port relative to a clock edge.

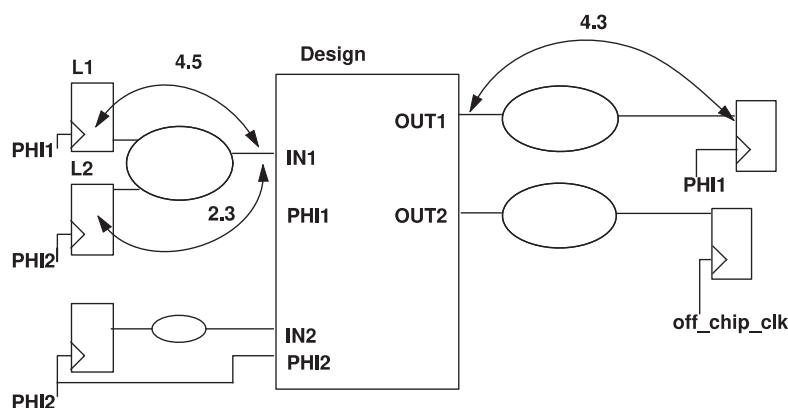


Figure B.2: Two-Phase Clocking Example for Setting Port Delays (from [76]).

Figure B.2 is used for the following examples.

Example 1

If the delay from L1 clock port to IN1 is 4.5, this `set_input_delay` command applies:

```
1 prompt > set_input_delay 4.5 -clock PHI1 {IN1}
```

Example 2

If paths from multiple clocks or edges reach at the same port, specify each one using the `-add_delay` option.

```
1 prompt > set_input_delay 2.3 -clock PHI2 -add_delay {IN1}
```

Example 3

This example shows a 7 ns input delay defined relative to clock CLK1. The command for this delay is:

```
1 prompt > set_input_delay 7 IN1 -clock CLK1
```

In contrast, the `set_output_delay` command specifies the output delays. An output delay represents an external timing path from an output port to a register. The maximum output delay value should be equal to the length of the longest path to the register data pin, plus the setup time of the register. The minimum output delay value should be equal to the length of the shortest path to the register data pin, minus the hold time. Analyze Figure B.2 for the following examples.

Example 1

To set an output delay of 4.3 relative to the rising edge of clock port PHI1 on port OUT1.

```
1 prompt > set_output_delay 4.3 -clock PHI1 {OUT1}
```

Example 2

If the library hold time is 0.7, the minimum output delay is the minimum path length minus the library hold time ($3.5 - 0.7$).

```
1 prompt > set_input_delay 2.8 -min -rise -clock PHI1 {OUT1}
```

B.2.1 Disable Timing

The `set_disable_timing` is a command used to exclude specific timing paths from timing analysis. This command is useful for situations where a path should not be considered valid for setup or hold checks, disabling timing from clocks that shouldn't propagate, or avoiding Design Rule Checks (DRC) violations.

```
1 prompt > set_disable_timing -from [get_pins U1/A] -to [get_pins U1/Y]
```

However, disabling a valid timing path can mask actual timing problems or violations, potentially leading to design failure in hardware.