UC Davis

UC Davis Electronic Theses and Dissertations

Title

RL-CHIP-Bench: A Benchmark-Integrated Framework for Sequential Chip Design Optimization with Real EDA Feedback

Permalink

https://escholarship.org/uc/item/91r719ss

ISBN

9798193263142

Author

Liu, Yichen

Publication Date

2026-09-08

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

RL-CHIP-Bench: A Benchmark-Integrated Framework for Sequential Chip Design Optimization with Real EDA Feedback

By

YICHEN LIU

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Houman Homayoun, Chair

Anthony Thomas

Hussain Al-Asaad

Committee in Charge

2026

Abstract

RL-CHIP-Bench: A Benchmark-Integrated Framework for Sequential Chip Design Optimization with Real EDA Feedback

by

YICHEN LIU

MASTER OF SCIENCE in Electrical and Computer Engineering

University of California, Davis

Houman Homayoun, Chair

Modern digital chip design requires repeated optimization decisions across synthesis, timing analysis, and physical implementation. Although reinforcement learning and other machine-learning methods provide promising approaches for exploring large design spaces, applying them to electronic design automation (EDA) requires more than an optimization policy. A practical learning-based system must also provide a stable agent–environment interface,

valid design transformations, reproducible task definitions, real backend evaluation, and structured experimental outputs.

This thesis presents RL-CHIP-Bench, a benchmark-integrated framework for sequential chip-design optimization with real EDA feedback. The framework separates benchmark control, agent interaction, semantic action representation, backend execution, quality-of-result (QoR) evaluation, reward computation, and artifact logging into modular components. A reinforcement-learning-compatible environment exposes a standardized sequential interface, while a fixed-dimensional semantic action representation maps agent decisions to design-dependent circuit transformations. Benchmark manifests define design resources, allowed actions, episode constraints, and evaluation settings independently of the agent.

The current validated implementation connects these components to an OpenROAD backend. Semantic actions are translated into gate-level netlist modifications, after which OpenROAD is used to obtain timing and other available QoR measurements. A reasoning-based controller demonstrates that candidate actions can also be evaluated through temporary execution and state restoration while using the same environment and backend interfaces intended for future reinforcement-learning agents.

Experimental validation demonstrates an operational end-to-end path from benchmark task loading and agent decision making to real netlist modification, OpenROAD evaluation, reward feedback, and structured result generation. The available QoR measurements confirm

that the framework reports both beneficial and unfavorable design effects rather than treating successful execution as optimization success. In the current evaluation sequence, timing remains feasible but the positive timing margin is reduced and power increases slightly, indicating that the tested transformation sequence does not yet provide strong QoR improvement.

The primary contribution of this work is therefore the design and validation of reusable research infrastructure rather than a claim of state-of-the-art timing optimization. The current implementation provides a foundation for future work involving trained reinforcement-learning policies, graph-based observations, improved target selection, additional circuit transformations, and evaluation across broader benchmark suites.

Contents

Contents **v**

List of Figures **vii**

List of Tables **viii**

1 Introduction **1**
- 1.1 Motivation and Research Problem 1
- 1.2 Research Scope and System Objectives 4
- 1.3 System Architecture Overview 5
- 1.4 Contributions 8
- 1.5 Thesis Organization 10

2 Background **12**
- 2.1 Reinforcement Learning for Sequential Optimization 12
- 2.2 Digital Design Optimization and Quality of Result 20
- 2.3 Graph Representation of Digital Circuits 27
- 2.4 Machine Learning for EDA Infrastructure and Benchmarking 33
- 2.5 Chapter Summary 41

3 RL-CHIP-Bench: System Design and Implementation **43**
- 3.1 Research Methodology and System Scope 43
- 3.2 RL-CHIP-Bench Architecture 46
- 3.3 RL-Compatible Environment and Sequential Optimization Interface 52
- 3.4 Semantic Action Representation and Target Mapping 62
- 3.5 Benchmark Harness and Task Specification 70
- 3.6 Backend Execution and OpenROAD-Based QoR Evaluation 78
- 3.7 Reasoning-Based Agent and Candidate Evaluation 87
- 3.8 Experimental Setup and Validation 95
- 3.9 Implementation Challenges and Limitations 105
- 3.10 Chapter Summary 113

4 Conclusion **116**
4.1 Summary of the Work 116
4.2 Main Contributions 117
4.3 Experimental Findings 119
4.4 Limitations 120
4.5 Future Work 122
4.6 Final Remarks 124

Bibliography **126**

List of Figures

1.1 The current validated implementation uses a reasoning-based agent, while the same interface is designed to support future reinforcement-learning and heuristic agents. 6

3.1 Implementation architecture of RL-CHIP-Bench. The benchmark harness constructs the task and runtime environment, the environment converts agent decisions into semantic actions, the backend runner applies design transformations and invokes OpenROAD, and the resulting QoR measurements are converted into benchmark feedback. The agent is separated from backend execution so that different decision-making methods can reuse the same infrastructure. 47
3.2 Execution flow of one RL-CHIP-Bench environment step. A submitted action is decoded and checked against task-specific restrictions before backend execution. Valid actions are evaluated using the real EDA backend, while invalid or disallowed actions are rejected without modifying the design. The resulting QoR measurements are converted into the next observation and reward. 55
3.3 Mapping from a fixed-dimensional discrete agent action to a concrete design transformation. The action components are integer indices whose configured ranges define the agent-facing action space. The target-bin index is then resolved to a design-dependent circuit object before backend execution. 64
3.4 Benchmark control flow in RL-CHIP-Bench. A task manifest is converted into a structured runtime specification, validated before execution, and used to construct the environment and backend. The harness then executes the selected submission and records structured artifacts and benchmark outputs. 73
3.5 Backend execution path in the current RL-CHIP-Bench implementation. A semantic action is converted into a concrete netlist modification, evaluated through OpenROAD, and parsed into updated QoR measurements that are returned to the environment. 82
3.6 Reasoning-based candidate evaluation in the current RL-CHIP-Bench submission. Candidate semantic actions can be temporarily evaluated using the same environment and backend used for normal execution. State restoration allows multiple candidate actions to be compared before the selected action is committed. 90

List of Tables

2.1 Mapping between reinforcement learning concepts and chip optimization. 19

3.1 Configuration of the current validated RL-CHIP-Bench task. 78
3.2 Main components of the current validated backend execution path. 86
3.3 Experimental configuration of the current validated RL-CHIP-Bench run. 97
3.4 Validation status of the current RL-CHIP-Bench execution path. 100
3.5 Measured QoR values from the available OpenROAD-backed evaluation sequence. 101
3.6 QoR progression across the available OpenROAD evaluation records. 102
3.7 Major implementation challenges and corresponding design responses. 113

Acknowledgments

I am deeply grateful to my advisor, Professor Houman Homayoun, for his supervision, encouragement, and support throughout my graduate study and research at UC Davis. His guidance provided an important academic foundation for this work and helped me develop the project from an engineering prototype into a more complete research system.

I also sincerely thank Kevin for his guidance and support throughout my laboratory research. His advice was especially helpful in understanding the project as more than a coding task, including EDA tool integration, benchmark design, debugging, experimental validation, and practical engineering trade-offs.

I would also like to thank my committee members, Professor Anthony Thomas and Professor Hussain Al-Asaad, for their time, careful feedback, and evaluation of this thesis. Their comments helped improve the focus, technical organization, and clarity of the final document.

Finally, I would like to thank my lab members, classmates, friends, and family for their support throughout my master's study. Their encouragement helped me through the many iterations of debugging, system integration, experimentation, and thesis writing required to complete this work.

ChatGPT, a large language model, was used with review and input by the author to assist with organizing the thesis structure and refining the presentation and development of ideas.

Chapter 1

Introduction

1.1 Motivation and Research Problem

Modern integrated-circuit design relies on a long sequence of electronic design automation (EDA) stages, including logic synthesis, technology mapping, placement, routing, timing analysis, and physical optimization. Each stage transforms the design while attempting to satisfy constraints on performance, power, area, signal integrity, and manufacturability [16, 10]. As circuit complexity increases, these stages involve a large number of discrete and interdependent decisions. A local change that improves one design metric may degrade another, and its effect may also alter which transformations are useful later in the flow.

Timing optimization is a representative example of this difficulty. Common transformations include buffer insertion, gate resizing, threshold-voltage replacement, gate cloning, and local logic restructuring. These operations affect delay through changes in drive strength,

load capacitance, signal slew, and circuit structure. Gate sizing and buffer insertion are particularly coupled: optimizing them independently can restrict the available solution space or produce locally reasonable but globally inefficient choices [4]. As a result, design optimization is not simply a one-time prediction problem. It can be viewed as a sequential process in which each action changes the design state and influences the value of future actions.

Machine learning has become an increasingly active direction for addressing these large design spaces. Prior work has applied reinforcement learning to chip placement, gate sizing, and engineering-change-order timing repair [12, 11, 9, 8]. In reinforcement learning, an agent repeatedly observes the current state of a system, selects an action, and receives a reward that measures the result of that action [15]. This interaction model is relevant to chip optimization because an optimization agent can observe design-quality measurements, apply a circuit transformation, and then evaluate whether the modified design has improved.

However, applying reinforcement learning to EDA requires more than selecting a learning algorithm. The agent must interact with an environment that can represent a real design, restrict actions to valid circuit transformations, execute those transformations, and return reliable feedback. In this work, backend evaluation is intentionally connected to an actual EDA flow rather than only to a manually defined proxy. This improves the fidelity of the feedback because actions are evaluated under real design resources, timing constraints, and tool behavior. The trade-off is that each interaction becomes substantially more expensive, introducing practical issues involving tool invocation, report parsing, failure handling, and sample efficiency. In addition, experiments must be recorded under a consistent task and

scoring protocol if results from different agents are to be compared meaningfully.

These infrastructure requirements remain an important challenge in machine-learning-driven EDA research. Existing work has identified limited standardization, fragmented tools and datasets, incompatible interfaces, and incomplete experimental reporting as major barriers to reproducibility [2]. Open-source infrastructure such as OpenROAD has made it possible to execute realistic digital-design flows without relying entirely on proprietary tools, while related work on OpenROAD and CircuitOps has shown how Python-based interfaces and machine-learning-oriented data representations can connect EDA engines with learning workflows [3]. Nevertheless, an optimization framework must still define how tasks, agents, actions, backend execution, rewards, scores, and output artifacts fit together as one reproducible system.

This thesis addresses that system-level problem through RL-CHIP-Bench. The system provides a common execution and evaluation structure for sequential chip-design optimization. The framework is designed around a standardized agent–environment interaction model, a fixed semantic action representation, a benchmark control layer, and a real OpenROAD-based evaluation path. Its purpose is to provide a common interface through which reinforcement-learning agents, heuristic controllers, and reasoning-based agents can interact with chip-design tasks under the same execution and logging protocol.

The current work is primarily an infrastructure and system-integration contribution. It demonstrates an operational path from benchmark task definition to circuit transformation, real backend evaluation, reward computation, and structured result generation. The current

agent does not yet constitute a mature timing optimizer, and the experimental scope does not establish state-of-the-art quality-of-result improvement. Instead, the main research question is how to construct a reliable and extensible environment in which stronger optimization agents can later be developed, evaluated, and compared.

1.2 Research Scope and System Objectives

The objective of this research is not to propose a new reinforcement learning algorithm, nor to outperform existing optimization methods on a particular benchmark. Instead, this work focuses on the software infrastructure required to connect machine-learning methods with realistic chip-design workflows.

A practical reinforcement-learning environment for EDA requires substantially more than a collection of optimization algorithms. It must provide a consistent representation of design states, define valid optimization actions, execute those actions through an EDA backend, evaluate the resulting design quality, and return standardized observations and rewards to the learning agent. In addition, the environment should support reproducible experiments so that different optimization strategies can be evaluated under identical conditions [2].

To address these requirements, RL-CHIP-Bench separates task definition, environment construction, backend execution, and agent implementation into independent components. Different decision-making methods can therefore reuse the same benchmark infrastructure without redefining the underlying execution flow.

The environment follows the interaction model commonly adopted by reinforcement learning, where an agent repeatedly observes the current system state, selects an optimization action, receives feedback from the environment, and continues until a stopping condition is reached [15]. Backend execution in the current implementation is provided by OpenROAD, an open-source digital-design framework with timing-analysis and physical-design capabilities [3]. The role of OpenROAD within the RL-CHIP-Bench execution path is described in detail in Chapter 3.

Although the current implementation focuses on representative timing-related circuit transformations, including buffer insertion, the proposed framework is intentionally designed to be extensible. Additional optimization actions, benchmark tasks, and learning algorithms can be incorporated without changing the overall system architecture. This modularity makes the framework suitable not only for reinforcement learning, but also for other decision-making approaches such as heuristic optimization, graph-based methods, and reasoning-based agents.

1.3 System Architecture Overview

The environment adopts the Gymnasium application programming interface, which standardizes reinforcement-learning tasks through functions such as `reset()` and `step()`.

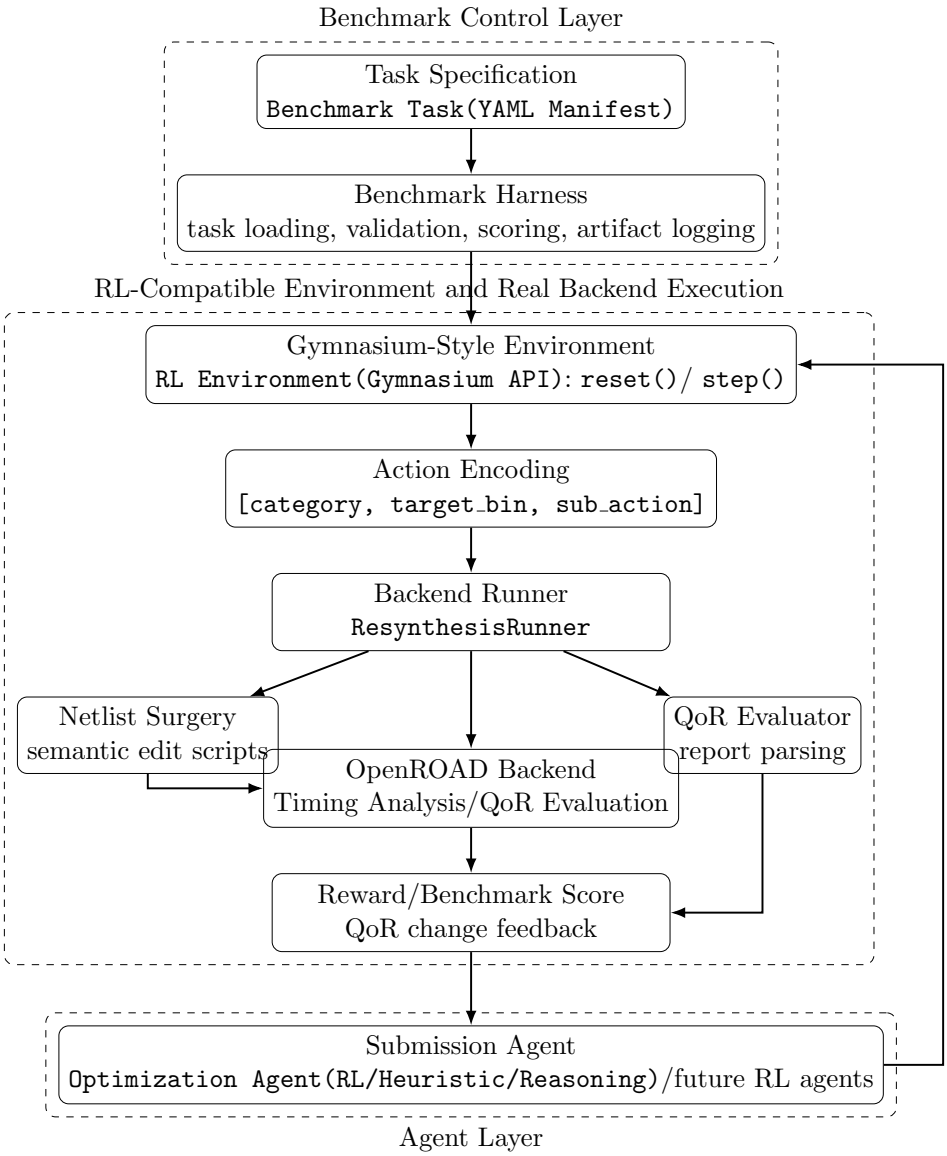

Figure 1.1: The current validated implementation uses a reasoning-based agent, while the same interface is designed to support future reinforcement-learning and heuristic agents.

Figure 1.1 illustrates the overall architecture of the proposed framework. Rather than treating reinforcement learning as an isolated optimization algorithm, the framework or-

ganizes the complete optimization workflow into several independent but connected layers. This modular organization separates benchmark definition, environment execution, backend evaluation, and decision making, allowing each component to evolve independently while maintaining a consistent experimental interface.

The optimization process begins with a benchmark task, which specifies the circuit, technology library, design constraints, and evaluation objectives. A benchmark harness then prepares the execution environment by loading the required design files and constructing the optimization task according to a standardized interface. This design allows different benchmark instances to share the same execution pipeline while varying only task-specific configurations.

The benchmark harness interacts with a reinforcement-learning environment that exposes chip optimization as a sequential decision-making problem. At each optimization step, the environment provides an observation describing the current design state, accepts an optimization action from the agent, executes the corresponding circuit transformation, and evaluates the updated design before returning a new observation together with a reward. This interaction follows the standard reinforcement-learning paradigm while remaining independent of the specific learning algorithm [15].

The environment delegates design execution and QoR measurement to an OpenROAD-based backend. This backend receives the decoded circuit transformation, evaluates the modified design, and returns the resulting measurements to the environment. The detailed execution path is presented in Section 3.6.

The backend produces timing reports and other design-quality measurements, which are converted into standardized observations and reward values for the optimization agent. Because this evaluation interface is independent of the decision-making algorithm, different optimization strategies—including reinforcement-learning agents, heuristic methods, graph-based models, and reasoning-based approaches—can be evaluated under the same benchmark protocol without modifying the underlying execution pipeline.

This modular architecture forms the foundation of the remainder of this thesis. Subsequent chapters describe the technical background supporting this design, followed by the implementation of the framework and the experiments conducted using the proposed benchmark infrastructure.

1.4 Contributions

The main contribution of this thesis is the design and implementation of a modular benchmark infrastructure for chip optimization using real EDA feedback. Rather than focusing only on one optimization algorithm, the work develops the surrounding system needed to define tasks, execute circuit transformations, evaluate design quality, and compare different decision-making methods under a common interface.

The specific contributions are summarized as follows:

- **A benchmark-oriented architecture for chip optimization.** The framework separates task specification, benchmark control, environment interaction, backend execu-

tion, scoring, and agent implementation into distinct layers. This organization reduces dependence on manually configured scripts and provides a clearer path for reproducing and comparing optimization experiments.

- **An RL-compatible environment abstraction.** Chip optimization is exposed as a sequential interaction process in which an agent observes the current design state, selects a circuit transformation, and receives quality-of-result feedback after backend evaluation. The environment uses a fixed action representation and standardized observation, reward, and episode interfaces, allowing different learning or reasoning methods to interact with the same task.

- **A semantic action layer for design-dependent circuit transformations.** The framework separates compact agent actions from concrete netlist edits. This action abstraction allows a stable interface to be maintained even when circuit instances and available optimization targets vary across designs.

- **Integration with an OpenROAD backend.** Candidate actions are translated into concrete netlist modifications and evaluated through OpenROAD-based timing analysis. The resulting quality-of-result metrics are returned to the environment and preserved for reward computation and benchmark analysis.

- **A unified interface for multiple agent types.** The environment and benchmark harness are independent of the decision-making method. The current implementation

supports a reasoning-based submission, while the same interface is intended to support future reinforcement-learning, heuristic, and graph-based agents.

- **Structured experiment logging and reproducible outputs.** Benchmark runs generate machine-readable summaries and final metrics instead of relying only on console output or screenshots. This makes individual runs easier to inspect and provides the foundation for future comparisons across agents and tasks.

The current contribution is therefore strongest in framework architecture, benchmark integration, environment design, and real-backend execution. The present experiments validate that these components can operate together in an end-to-end flow, but they do not yet demonstrate consistently strong timing improvement across a broad benchmark suite. Improving optimization quality, expanding task coverage, and evaluating trained learning agents remain important directions for future work.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 introduces the technical background required to understand the proposed framework. It reviews reinforcement learning for sequential optimization, graph-based representations of digital circuits, timing-driven chip optimization, and benchmark infrastructure for machine-learning-driven EDA. The chapter focuses only on concepts that directly support the research presented later in the thesis.

Chapter 3 presents the RL-CHIP-Bench framework in detail. It describes the benchmark harness, the reinforcement-learning-compatible environment, semantic action encoding, backend netlist transformation, OpenROAD-based quality-of-result evaluation, and the current reasoning-based submission. It also presents the validated experimental setup, implementation challenges, current system status, and limitations.

Chapter 4 concludes the thesis by summarizing the main technical contributions and discussing future directions, including broader benchmark validation, stronger action selection methods, graph-enhanced observations, learned surrogate models, and trained reinforcement-learning agents.

Chapter 2

Background

This chapter introduces the technical foundations required to understand the RL-CHIP-Bench framework. The discussion focuses on reinforcement learning as a sequential optimization method, graph-based representations of digital circuits, timing-driven design optimization, and reproducible benchmark infrastructure for machine-learning-driven electronic design automation. These topics provide the conceptual basis for the environment, action, reward, backend, and evaluation interfaces described in Chapter 3.

2.1 Reinforcement Learning for Sequential Optimization

Reinforcement learning (RL) addresses sequential decision-making problems in which an agent improves its behavior through interaction with an environment. At each time step,

the agent observes information about the current state, selects an action, receives a reward, and transitions to a new state. Unlike supervised learning, the agent is not provided with a correct action label for every state. Instead, it must determine which actions are useful from the rewards generated by its interactions [15].

This interaction model is suitable for optimization problems in which a solution is constructed through a sequence of dependent decisions. A locally beneficial action may change the system in a way that affects the value of later actions. The objective is therefore not only to maximize the immediate improvement produced by one action, but to select a sequence of actions that produces a high cumulative return.

Markov Decision Process Formulation

A reinforcement learning problem is commonly modeled as a Markov Decision Process (MDP). An MDP can be represented by the tuple

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)\,, \tag{2.1}$$

where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P(s' \mid s, a)$ defines the transition dynamics, $R(s, a)$ defines the reward associated with an action, and $\gamma \in [0, 1]$ is the discount factor [15]. The discount factor controls the relative importance of future rewards.

Starting from time step t, the discounted return is defined as

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.2}$$

where r_{t+k+1} is the reward received after a future environment transition. The agent seeks a policy $\pi(a \mid s)$ that maximizes the expected return.

The Markov property assumes that the current state contains sufficient information to describe the future evolution of the system. In practical engineering systems, this assumption may only be approximate because the observation may omit internal variables or historical information. Nevertheless, the MDP formulation remains useful because it provides a structured way to define observations, actions, rewards, and episode behavior.

Policies and Value Functions

A policy describes how the agent selects actions. A deterministic policy maps a state directly to an action, while a stochastic policy defines a probability distribution over the available actions. The state-value function measures the expected return from a state under a policy:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s\right]. \tag{2.3}$$

Similarly, the action-value function evaluates the expected return after selecting action a in state s and then following policy π:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s, A_t = a\right]. \tag{2.4}$$

Value functions are important in sequential optimization because the immediate reward may not reflect the complete effect of an action. An action can produce a small short-term improvement while creating a more favorable state for later optimization, or it can produce an immediate gain that reduces future flexibility.

Value-Based Reinforcement Learning

Q-learning is a model-free value-based method that learns the optimal action-value function from sampled transitions [17]. Its tabular update is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right], \tag{2.5}$$

where α is the learning rate. Because the method is model-free, it does not require an explicit analytical model of the environment dynamics.

For large or continuous state spaces, maintaining a separate table entry for every state–action pair is not practical. Deep Q-Networks (DQN) replace the tabular action-value representation with a neural network $Q(s, a; \theta)$ parameterized by θ [13]. DQN uses experience replay to store transitions and samples mini-batches from this replay memory during training. A target network is also used to reduce instability caused by rapidly changing bootstrap targets.

For a transition $(s_t, a_t, r_{t+1}, s_{t+1})$, a common DQN target is

$$y_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-), \tag{2.6}$$

where θ^- represents the target-network parameters. The online network is trained by minimizing

$$\mathcal{L}(\theta) = \left(y_t - Q(s_t, a_t; \theta)\right)^2. \tag{2.7}$$

Value-based methods are especially convenient when the action space is discrete. However, their effectiveness depends on the quality of the state representation, the validity of the action set, and the cost of collecting transitions.

Policy-Gradient and Actor–Critic Methods

Policy-gradient methods directly optimize a parameterized policy rather than deriving it only from an action-value function. These methods are useful when the policy is stochastic or when the action space is difficult to handle through explicit maximization.

Proximal Policy Optimization (PPO) is a widely used policy-gradient method that limits the size of policy updates through a clipped optimization objective [14]. The clipping mechanism reduces destructive updates while retaining a relatively simple implementation. Actor–critic methods combine a policy model, called the actor, with a value estimator, called the critic. The actor selects actions, while the critic estimates the expected return and provides a lower-variance learning signal.

Although the current validated RL-CHIP-Bench submission is not trained using DQN or PPO, these algorithm families motivate the environment design. A reusable optimization environment should expose observations, actions, scalar rewards, and episode boundaries in a form that can support value-based, policy-gradient, or actor–critic methods without modifying the backend execution logic.

Exploration, Sample Efficiency, and Expensive Environments

A central challenge in reinforcement learning is balancing exploration and exploitation. Exploration allows an agent to discover actions whose values are not yet known, while exploitation favors actions that currently appear beneficial. In inexpensive simulated environments, large numbers of transitions can often be generated at relatively low cost. This assumption does not necessarily hold in EDA optimization.

When one environment interaction requires modifying a circuit, invoking an EDA tool, performing timing analysis, and recomputing quality-of-result metrics, collecting training transitions can become substantially more expensive. Prior learning-based EDA work has addressed this problem in different ways. Reinforcement-learning-based chip placement has used faster approximate reward signals because evaluating true physical-design objectives with industrial EDA tools can be extremely costly [12]. RL-Sizer reduces timing-evaluation overhead by using changes in a local graph neighborhood rather than performing a full-netlist timing update for every individual sizing decision [9]. IR-aware ECO optimization

also identifies sample efficiency as an important limitation of RL and uses problem-specific constraints and domain knowledge to reduce the effective action and search spaces [8].

These approaches illustrate a general trade-off in learning-based EDA. Higher-fidelity backend evaluation provides more reliable information about the effect of a design transformation, but repeated tool-backed evaluations can limit the amount of exploration that is practical. Sample efficiency therefore becomes closely related to action-space restriction, candidate filtering, approximation, and reuse of previously collected experience.

This trade-off is directly relevant to the implementation presented in Chapter 3. Section 3.6 describes the limited mechanisms currently used in RL-CHIP-Bench to avoid some unnecessary backend evaluations, while Section 3.7 explains how the reasoning-based controller evaluates a restricted set of candidate actions using state restoration. These mechanisms mitigate evaluation cost but do not eliminate the broader sample-efficiency problem.

Mapping Reinforcement Learning to Chip Optimization

Chip optimization can be expressed using the same agent–environment structure. The state represents the current condition of the design, the action represents a circuit transformation, and the reward measures the change in quality of result. A simplified mapping is shown in Table 2.1.

Prior work has demonstrated this formulation in several EDA domains. Chip placement has been modeled as a sequential decision process in which an agent places circuit components

Table 2.1: Mapping between reinforcement learning concepts and chip optimization.

RL concept	**Chip optimization interpretation**
State or observation	Timing and design-quality information describing the current circuit.
Action	A semantic transformation such as buffer insertion, gate resizing, or local logic restructuring.
Environment	The design-processing flow that applies an action and evaluates the modified circuit.
Reward	A scalar value derived from changes in timing, power, wirelength, or other QoR metrics.
Episode	A bounded sequence of optimization steps applied to one benchmark task.
Policy	The decision rule used to select the next circuit transformation.

onto a chip canvas [12, 11]. RL-Sizer applies reinforcement learning to gate sizing for timing optimization [9], while later work uses RL for IR-aware engineering-change-order timing repair [8]. These studies show that RL can support discrete EDA optimization, but they also demonstrate that each application requires a carefully designed state representation, action space, reward function, and tool interface.

In RL-CHIP-Bench, reinforcement learning is used primarily as an environment-design abstraction. The framework defines a sequential interaction protocol through which an agent can observe QoR information, submit a semantic circuit action, invoke real backend evaluation, and receive reward feedback. The current reasoning-based agent operates on this same interface, demonstrating that the environment abstraction can support decision-making methods beyond a single RL algorithm.

2.2 Digital Design Optimization and Quality of Result

Digital integrated-circuit implementation transforms a behavioral hardware description into a manufacturable physical design. A typical application-specific integrated circuit (ASIC) flow includes register-transfer-level design, logic synthesis, technology mapping, floorplanning, placement, clock-tree synthesis, routing, timing analysis, and physical verification [16, 10]. Although these stages are often presented as a sequence, they are strongly coupled. Decisions made during synthesis affect the structure presented to placement, while placement and routing determine interconnect delays that may require later logic or physical optimization.

The objective of the implementation flow is not to optimize a single metric independently. Instead, the design must satisfy functional and timing constraints while balancing power, performance, and area. These objectives are commonly summarized as the power, performance, and area (PPA) trade-off. Improving one component may degrade another. For example, increasing gate drive strength can reduce delay but also increase area, capacitance, and power. Similarly, inserting buffers can improve signal transition and reduce the load seen by a driver, but additional cells and interconnect may increase power, area, or routing demand.

This multi-objective and design-dependent behavior motivates the use of quality-of-result (QoR) metrics. QoR refers to the measurable properties used to evaluate the outcome of an EDA flow or optimization step. In the context of this thesis, the most important QoR signals

are timing metrics, particularly worst negative slack and total negative slack, together with secondary measurements such as power and wirelength.

Static Timing Analysis

Static timing analysis (STA) evaluates whether signal propagation through a digital circuit satisfies the timing constraints imposed by the clock and interface specifications. Unlike dynamic simulation, STA does not require explicit input vectors. Instead, it analyzes timing paths using cell-delay models, interconnect information, clock definitions, and timing constraints.

For a timing endpoint, slack is defined as the difference between the required arrival time and the actual arrival time:

$$\text{slack} = t_{\text{required}} - t_{\text{arrival}}. \tag{2.8}$$

A nonnegative slack indicates that the endpoint satisfies its timing requirement. A negative slack indicates a timing violation because the signal arrives later than permitted.

Worst negative slack (WNS) is the smallest slack among all analyzed endpoints:

$$\text{WNS} = \min_{e \in \mathcal{E}} \text{slack}(e), \tag{2.9}$$

where $\mathcal{E}$ is the set of timing endpoints. WNS describes the most severe timing violation in the design. An improvement in WNS means that the worst timing path has moved closer

to, or beyond, timing closure.

Total negative slack (TNS) accumulates the negative slack over all violating endpoints:

$$\mathrm{TNS} = \sum_{e \in \mathcal{E}} \min\left(0, \mathrm{slack}(e)\right). \tag{2.10}$$

Depending on the reporting convention, TNS may be presented as a negative quantity or as the magnitude of the accumulated violations. This thesis follows the convention produced by the backend tool and interprets improvement according to the corresponding before-and-after change. While WNS measures the single worst violation, TNS provides information about the broader distribution of timing failures. A design may improve its worst path while leaving many other violating endpoints unchanged; therefore, both metrics are useful when evaluating an optimization action.

Timing Closure as an Iterative Optimization Process

Timing closure is the process of modifying a design until all required timing constraints are satisfied. It is typically iterative because correcting one path can affect other parts of the circuit. Changes in gate size, fanout, placement, or routing can alter capacitance, slew, and delay along multiple paths.

A simplified timing-optimization loop consists of the following steps:

1. analyze the current design and identify critical or violating paths;

2. select one or more candidate circuit transformations;

3. apply the selected transformation;

4. rerun timing analysis and collect updated QoR metrics;

5. retain, reject, or revise the transformation based on its measured effect.

This loop resembles sequential decision making because every accepted transformation changes the design presented to the next iteration. The best action cannot always be selected from its immediate local effect alone. An action may remove one violation while creating a new critical path, or it may produce a small current improvement that enables a more useful later transformation.

Buffer Insertion

Buffer insertion introduces one or more buffer cells into a net to divide a large electrical load, improve transition time, or reduce the delay observed by an upstream driver. A buffer can isolate a weak source from a large fanout or long interconnect, allowing the signal to be driven in multiple stages.

The effectiveness of buffer insertion depends on several factors, including the original driver strength, load capacitance, wire resistance, buffer type, and insertion location. Inserting a buffer at an unsuitable location may provide little timing benefit or may even increase delay. The operation also adds cells and nets, which can affect area, power, and routing congestion.

Buffer insertion is therefore not a simple binary decision. A complete decision may need to determine whether a buffer should be inserted, which cell size should be used, and where the new stage should be placed in the logical or physical path. These choices create a discrete action space that depends on both the circuit structure and the available standard-cell library.

Gate Sizing

Gate sizing changes the drive strength of a standard-cell instance by replacing it with a functionally equivalent cell from the same library family. Larger cells generally provide stronger drive and lower delay for a given load, but they also increase input capacitance, area, and power. Downsizing a cell can reduce power and area, but may degrade timing.

Gate sizing is combinatorial because each optimizable instance may have several discrete library alternatives. For a design with many cells, the number of possible assignments grows rapidly. RL-Sizer demonstrates one learning-based formulation in which gate sizing is treated as a sequential control problem, with graph-based state features, discrete sizing actions, and timing improvement as the optimization objective [9].

Late-stage gate sizing may also be subject to additional physical constraints. In IR-aware engineering-change-order optimization, for example, cell resizing is used to repair timing violations while limiting disturbance to an almost completed physical design [8]. This illustrates that an action that is legal at an early design stage may be unacceptable later because the allowable physical perturbation becomes much smaller.

Coupling Between Optimization Actions

Physical-design transformations cannot always be optimized independently. Buffer insertion and gate sizing are a clear example. A weak driver may require multiple buffers to drive a large load, while replacing the driver with an excessively large cell may improve its output delay but increase the load on the preceding stage. A balanced combination of moderate sizing and buffering may produce a better overall solution.

Recent work on differentiable physical optimization explicitly studies the interaction between buffer insertion and gate sizing, arguing that separate optimization can restrict the solution space and produce suboptimal resource allocation [4]. The interaction arises through capacitance, slew propagation, wire delay, and upstream loading. These dependencies reinforce the need to evaluate actions through the complete timing flow rather than estimating their value only from isolated local rules.

The broader action space considered by RL-CHIP-Bench follows the same motivation. Buffer operations, gate resizing, threshold-voltage replacement, cloning, and logic restructuring represent different ways to modify the timing behavior of a circuit. Although the current validated experiment uses a narrower subset, the framework separates action representation from backend execution so that additional transformations can be introduced under the same environment interface.

QoR-Based Reward Design

When chip optimization is formulated as a reinforcement-learning problem, reward design provides the connection between measured design quality and the learning objective. A reward may be derived from changes in timing, power, area, wirelength, or other quality-of-result metrics, depending on the optimization task.

Timing-oriented EDA applications commonly give particular importance to slack-related metrics. For example, RL-Sizer formulates gate sizing as a sequential optimization problem in which timing improvement provides the primary optimization signal [9]. IR-aware ECO optimization similarly combines timing repair with additional physical-design constraints when defining the learning problem [8].

Reward design becomes more complicated when several QoR objectives must be considered simultaneously. A change that improves timing may increase power or area, while an action that reduces a secondary cost may be unacceptable if it introduces a timing violation. Consequently, the relative importance of different metrics can depend on the current design condition and the objective of the optimization stage.

A scalar reward also does not replace the underlying QoR measurements. Two actions may produce similar scalar rewards while affecting timing and power in different ways. For this reason, learning-based EDA experiments should report the original measured metrics in addition to any scalar reward used by the agent.

The specific observation vector and reward logic used by RL-CHIP-Bench are part of the

implementation developed in this thesis and are therefore described in Section 3.3 rather than in this background chapter.

Real Backend Evaluation

Analytical approximations and learned surrogate models can reduce evaluation cost, but their predictions may differ from the result produced by a complete EDA flow. Tool-level effects can depend on technology libraries, timing constraints, circuit structure, and the specific implementation stage being evaluated.

Open-source infrastructure such as OpenROAD provides a way to evaluate learning-based EDA methods using executable design flows, while work combining OpenROAD and CircuitOps demonstrates how machine-learning systems can exchange structured information with EDA tools [3]. This creates a trade-off between evaluation fidelity and interaction cost: real tool-backed measurements provide stronger evidence of design impact, but they are generally more expensive to obtain than analytical or learned approximations.

The specific way in which RL-CHIP-Bench integrates OpenROAD is described in Section 3.6.

2.3 Graph Representation of Digital Circuits

Digital circuits are naturally structured as graphs. At the gate level, logic cells, ports, or other circuit objects can be represented as nodes, while signal dependencies and net

connections can be represented as edges. This graph-based view preserves relationships that are difficult to capture using only flat scalar features. In particular, timing and optimization behavior depend not only on the properties of an individual cell, but also on its local fan-in, fan-out, path position, and interaction with neighboring circuit elements.

A graph representation is therefore useful for machine-learning-driven EDA because it allows a model to reason about circuit structure directly. Instead of treating each gate as an independent sample, a graph model can aggregate information from nearby nodes and learn features that reflect both local context and broader topology.

Netlists as Graphs

A gate-level netlist can be represented as a directed graph

$$G = (V, E), \tag{2.11}$$

where V is the set of circuit objects and E is the set of directed signal connections. A node $v \in V$ may represent a logic gate, sequential element, input port, or output port. An edge $(u, v) \in E$ indicates that the output of object u drives an input of object v.

Each node can be associated with a feature vector

$$\mathbf{x}_v = \left[x_v^{(1)}, x_v^{(2)}, \ldots, x_v^{(d)}\right], \tag{2.12}$$

where the features may include cell type, fan-in, fan-out, logic depth, timing slack, ca-

pacitance, transition time, or other available design attributes. Edge features may also be included when net-level information such as wirelength, delay, or capacitance is available.

This representation is flexible because the same graph can support several downstream tasks. A node-level model may estimate whether a gate is important for timing, while a graph-level model may estimate overall design quality. A graph representation can also support target selection by identifying structurally similar regions or high-impact local neighborhoods.

Message Passing in Graph Neural Networks

Graph neural networks (GNNs) learn node representations by repeatedly combining a node's current features with information from its neighbors. A general message-passing update can be written as

$$\mathbf{h}_v^{(k)} = \phi^{(k)} \left(\mathbf{h}_v^{(k-1)}, \mathrm{AGG}^{(k)} \left(\{ \mathbf{h}_u^{(k-1)} \mid u \in \mathcal{N}(v) \} \right) \right), \tag{2.13}$$

where $\mathbf{h}_v^{(k)}$ is the representation of node v after layer k, $\mathcal{N}(v)$ is its neighborhood, $\mathrm{AGG}^{(k)}$ is a neighborhood aggregation function, and $\phi^{(k)}$ combines the aggregated neighborhood information with the node's own representation.

After several layers, the embedding of a node contains information from a multi-hop neighborhood. This is relevant to timing analysis because the effect of a local gate is often influenced by nearby drivers, loads, and path structure rather than by the gate alone.

GNNs have already been applied to gate-level circuit analysis. GNN-RE, for example, represents gates as graph nodes and uses learned structural information to identify module boundaries and classify sub-circuits in flattened gate-level netlists [1]. Although that task differs from timing optimization, it demonstrates that graph models can extract useful semantic information from circuit connectivity.

GraphSAGE for Inductive Node Representation

GraphSAGE is an inductive graph representation method that learns how to aggregate neighboring features instead of learning a separate embedding for every node [5]. This property is important when a model must operate on previously unseen graphs or designs.

A simplified GraphSAGE update can be written as

$$\mathbf{h}_{\mathcal{N}(v)}^{(k)} = \mathrm{AGG}^{(k)}\left(\left\{\mathbf{h}_u^{(k-1)} \mid u \in \mathcal{N}(v)\right\}\right) , \tag{2.14}$$

followed by

$$\mathbf{h}_v^{(k)} = \sigma\left(\mathbf{W}^{(k)}\left[\mathbf{h}_v^{(k-1)} \,\|\, \mathbf{h}_{\mathcal{N}(v)}^{(k)}\right]\right), \tag{2.15}$$

where $\|$ denotes concatenation, $\mathbf{W}^{(k)}$ is a learnable weight matrix, and σ is a nonlinear activation function.

The inductive design of GraphSAGE is useful for EDA research because benchmark tasks may involve different circuits with different graph sizes and topologies. A representation

method that generalizes across graphs is more appropriate than one that depends on a fixed set of node identities.

Potential Uses in Chip Optimization

Graph representations can support several parts of a chip optimization framework.

First, graph embeddings can enrich the state representation. Scalar QoR values such as WNS and TNS summarize global design quality, but they do not indicate where the timing problem is located or which local region is responsible. Node or subgraph embeddings can provide structural context that helps an agent distinguish between candidate targets.

Second, graph features can support action ranking. A learned model may estimate which cells or nets are more likely to produce useful timing improvement under a given transformation. This is particularly important when real backend evaluation is expensive, because evaluating every possible action through OpenROAD may be impractical.

Third, graph representations can support transfer across designs. If structurally similar circuit regions produce similar optimization responses, a model may reuse knowledge from previous designs. Recent work on Pieceformer studies this broader idea through scalable graph similarity learning and similarity-driven knowledge transfer for VLSI tasks [18]. Such transfer is relevant to future benchmark systems in which agents should generalize beyond one fixed design.

Finally, graph representations can support surrogate or partial models. A learned pre-

dictor could map local graph features and a candidate action to an estimated QoR change:

$$\hat{\Delta \mathbf{q}} = f_\theta \left(\mathbf{z}_{\text{graph}}, \mathbf{z}_{\text{target}}, a \right), \tag{2.16}$$

where $\mathbf{z}_{\text{graph}}$ represents global or regional circuit information, $\mathbf{z}_{\text{target}}$ represents the selected target, and a is the candidate action. Such a model does not need to replace the real backend. Instead, it can be used to filter or rank candidates before final verification through OpenROAD.

Limitations of Graph-Based Modeling

Graph learning also introduces several practical difficulties. Circuit graphs may contain large numbers of nodes and edges, making full-graph training expensive. The choice of graph construction can also affect the learned representation. For example, representing nets as edges is simple, but may lose information about multi-pin connectivity. Introducing explicit net nodes can preserve more structure but increases graph size.

Feature availability is another limitation. Structural features such as fan-in and fan-out are easy to obtain, while timing, physical, and technology-dependent features may require additional tool execution. A graph model trained only on connectivity may capture topology but miss the physical factors that determine actual delay.

There is also a distinction between representation quality and optimization quality. A well-separated embedding space does not automatically imply that the representation is

useful for selecting timing actions. The embedding must ultimately be validated through a downstream task such as QoR prediction, target ranking, or policy improvement.

For these reasons, graph learning is treated in this thesis as an important supporting direction rather than the main validated execution path. The current RL-CHIP-Bench environment uses a compact QoR-based observation for the real benchmark flow. Graph embeddings remain a natural extension for future agents and surrogate models, particularly when broader task coverage and more training data become available.

2.4 Machine Learning for EDA Infrastructure and Benchmarking

Machine-learning methods for electronic design automation are often evaluated through complex software pipelines rather than through isolated prediction models. A complete experiment may depend on circuit data, technology libraries, tool versions, timing constraints, feature extraction procedures, optimization scripts, agent implementations, and evaluation metrics. Differences in any of these components can change the final result, even when two studies use similar learning algorithms.

This creates a reproducibility challenge. A reported improvement may be difficult to interpret if the underlying task definition, tool configuration, action restrictions, or evaluation procedure is not fully specified. Recent work toward an ML EDA commons emphasizes the

need for shared standards, accessible infrastructure, consistent metrics, and reproducible workflows in machine-learning-driven EDA research [2].

For a benchmark framework, this means that the experimental conditions should be represented explicitly rather than being distributed implicitly across local scripts or agent implementations. Design files, technology configuration, timing constraints, allowed actions, backend settings, evaluation metrics, and stopping conditions should be recorded as part of the task definition or benchmark configuration. When these settings remain fixed while the agent changes, differences between runs are less likely to result from hidden changes in the experimental setup.

Role of a Benchmark Framework

A benchmark framework provides a controlled interface between the optimization method and the design task. Its purpose is not only to store input designs, but also to define how an experiment is initialized, which actions are legal, how the backend is executed, which metrics are collected, and how the final result is reported.

The reproducibility benefit comes from keeping these conditions independent of the agent implementation. For example, two agents evaluated on the same task can use the same netlist, technology library, timing constraints, action restrictions, backend configuration, episode budget, and scoring procedure. The decision-making method changes, while the experimental conditions remain fixed. This makes it easier to attribute differences in the

result to the agent rather than to hidden changes in configuration.

A practical benchmark for learning-based chip optimization should include at least the following components:

- a task specification describing the design, technology, constraints, and optimization objective;
- a validation layer that checks required files, configurations, and submission compatibility before execution;
- a standardized environment interface through which an agent observes the design state and submits actions;
- a backend interface that applies circuit transformations and evaluates the modified design;
- a scoring or reward mechanism derived from clearly defined QoR metrics;
- structured output artifacts that record execution status, actions, metrics, and final results.

Separating these responsibilities improves both experimental clarity and reproducibility. The agent determines how actions are selected, while the benchmark determines the task, legal action space, execution conditions, and evaluation procedure. As a result, a new decision-making method can be introduced without redefining the design flow or changing

the measurement protocol. This makes repeated runs easier to reproduce and reduces the risk that two agents are compared under unintentionally different experimental conditions.

Task Specification and Experimental Control

A task specification describes one optimization problem independently of a particular agent. In a chip-design setting, it may identify the circuit netlist, timing constraints, technology library, backend configuration, maximum episode length, available actions, and evaluation metrics.

This task-centered structure is important because EDA experiments are highly design-dependent. A transformation that is legal for one technology library may not be available in another. Similarly, the number and type of candidate circuit objects vary across designs. A benchmark task must therefore define not only the input circuit, but also the rules under which optimization is allowed.

Explicit task definitions also reduce hidden assumptions. Without a task manifest or equivalent configuration, important settings may be embedded inside local scripts or hard-coded file paths. Such experiments are difficult to reproduce because the effective task is distributed across several implementation files. A structured task description makes these dependencies easier to inspect, validate, and reuse across different submissions.

Standardized Environment Interfaces

A standardized environment interface separates the internal complexity of EDA execution from the decision-making logic of the agent. From the agent's perspective, the environment provides an observation, accepts an action, and returns a reward and updated state. Internally, however, one environment step may involve action decoding, legality checking, netlist modification, tool invocation, report parsing, and QoR calculation.

This separation follows the same motivation as general reinforcement-learning benchmarks. RLBench, for example, provides a shared task and environment interface for comparing robot-learning methods under consistent experimental conditions [7]. Open RL Benchmark similarly emphasizes tracked experiments, dependency information, and consistent reporting when comparing reinforcement-learning implementations [6].

Although these benchmarks operate in different application domains, the underlying principle is applicable to EDA: the environment and evaluation protocol should remain stable while the agent changes. This allows policy differences to be studied without conflating them with changes in backend execution or metric calculation.

Agent-Independent Evaluation

A useful benchmark should not assume that every submission uses the same learning algorithm. Different optimization methods may include value-based RL, policy-gradient methods, graph-based predictors, heuristic search, local optimization, or reasoning-based plan-

ning.

The benchmark should therefore define the interaction contract rather than the internal structure of the agent. An agent may use a neural network, a rule-based policy, or a search procedure, but it must submit actions through the same environment interface and receive the same backend-derived feedback.

This agent-independent design has two advantages. First, it supports fairer comparison because all methods operate under the same action restrictions and evaluation procedure. Second, it allows the benchmark infrastructure to remain useful as the decision-making method evolves. A new agent can be introduced without modifying the task definition, backend evaluator, or artifact format.

OpenROAD and CircuitOps as ML–EDA Infrastructure

OpenROAD provides an open-source digital-design implementation flow that includes major stages of synthesis and physical design. Its accessibility makes it suitable for research environments in which machine-learning methods must interact with real EDA tools rather than proprietary black-box flows.

Work on OpenROAD and CircuitOps demonstrates how EDA tools can be connected with machine-learning workflows through structured interfaces and circuit-level data representations [3]. CircuitOps provides graph-oriented circuit data that can support downstream learning tasks, while OpenROAD provides the execution backend needed to generate timing

and physical-design information.

The combination of data representation and tool execution is especially important for learning-based optimization. A model requires structured input features, but its decisions must ultimately be evaluated against the actual design flow. Infrastructure that connects these two layers reduces the gap between offline machine learning and tool-integrated EDA optimization.

In this thesis, OpenROAD is used primarily as the backend evaluator. The framework applies a circuit transformation, invokes the backend, and parses the resulting QoR measurements. This preserves a clear boundary between the agent and the EDA engine: the agent proposes a decision, while the backend determines its measured design impact.

Artifact Logging and Reproducibility

Console output alone is insufficient for reproducible experiments. Important information may be lost, truncated, or difficult to compare across runs. Structured artifacts provide a more reliable record of the execution process.

A benchmark run should preserve information such as:

- the task and agent configuration;
- the initial and final QoR metrics;
- the sequence of submitted actions;

- whether each action was valid and successfully executed;
- backend failures or warnings;
- the termination condition and total number of steps;
- the final benchmark score or cumulative reward.

Machine-readable outputs make it possible to compare runs programmatically and reduce the need to inspect screenshots or manually copy results from terminal logs. This is consistent with broader reproducibility goals in reinforcement learning and ML EDA [6, 2].

Artifact logging is also important for debugging. If an optimization result is unexpected, the recorded action sequence and intermediate QoR values can help determine whether the cause was poor agent behavior, an invalid transformation, backend failure, or metric-parsing error.

Benchmarking Beyond a Single Successful Run

A single end-to-end execution validates that the software components can interact, but it does not by itself establish optimization quality. A mature benchmark evaluation should include multiple tasks, multiple random seeds where applicable, baseline methods, repeated runs, and clearly defined aggregate metrics.

For learning-based methods, performance variation across seeds can be significant. For EDA tasks, variation may also arise from design structure, technology libraries, or backend

settings. Therefore, claims about optimization effectiveness should be based on repeated measurements rather than on one favorable run.

This distinction is important for the scope of the current work. RL-CHIP-Bench validates a real task-execution path that includes task loading, environment interaction, backend evaluation, and structured result generation. However, broader statistical comparison across agents and benchmark tasks remains future work. The current contribution is the infrastructure needed to support such experiments, rather than a claim that benchmark coverage and optimization performance are already complete.

2.5 Chapter Summary

This chapter introduced the technical foundations required for the design of RL-CHIP-Bench. Reinforcement learning provides the sequential decision-making abstraction used to define interactions among states, actions, rewards, and optimization episodes. Digital design optimization provides the practical problem setting, where local transformations such as buffer insertion and gate sizing must be evaluated through timing and other quality-of-result metrics. Graph representations provide a possible way to encode the structural information of gate-level circuits, while benchmark infrastructure provides the task control, reproducibility, and agent-independent evaluation needed for systematic experimentation.

Together, these concepts motivate a framework in which chip optimization is exposed through a standardized environment interface, while actual design changes are evaluated

through a real EDA backend. Chapter 3 presents the implementation of this framework, including its benchmark harness, environment design, semantic action abstraction, backend execution path, OpenROAD-based QoR evaluation, and current agent interface.

Chapter 3

RL-CHIP-Bench: System Design and Implementation

3.1 Research Methodology and System Scope

This chapter presents the detailed design and implementation of RL-CHIP-Bench. As discussed in Chapters 1 and 2, the central challenge is not only to define an optimization policy, but also to construct a reliable execution environment in which candidate circuit transformations can be represented, validated, applied, and evaluated using a real EDA backend.

The framework therefore treats chip optimization as a closed-loop engineering process. At each decision step, an agent receives information describing the current design state, selects a semantic circuit transformation, and submits the action through a standardized environment interface. The framework validates the action, maps it to a concrete design target,

applies the corresponding netlist transformation, invokes OpenROAD for quality-of-result (QoR) evaluation, and returns the measured result to the agent. This interaction follows the sequential decision-making formulation introduced in Chapter 2, while the implementation separates agent logic from benchmark control and backend execution.

The main research objective is to determine whether these components can be integrated into a modular and reproducible framework rather than implemented as a collection of design-specific scripts. This leads to four main system requirements.

1. **A stable agent–environment interface.** The decision-making method should interact with the chip-design task through a consistent observation, action, reward, and episode interface without directly controlling backend scripts.

2. **A design-independent action representation.** Because different circuits contain different numbers of instances and candidate optimization targets, the agent-facing action space should remain fixed while still being convertible into valid design-dependent transformations.

3. **Backend-derived QoR evaluation.** Candidate actions should be evaluated using QoR measurements produced by the configured EDA backend. OpenROAD provides this evaluation path in the current validated implementation [3].

4. **Benchmark-controlled execution and reproducibility.** Task configuration, allowed actions, resource validation, execution status, scoring, and output artifacts

should be controlled by the benchmark infrastructure so that different agents can later be evaluated under the same protocol [2, 6].

These requirements led to a layered implementation in which benchmark control, environment interaction, semantic action encoding, backend execution, QoR evaluation, and agent decision making are separated into distinct components. The architecture is described in Section 3.2.

The current validated implementation uses one benchmark task with an AES gate-level design implemented using the Nangate45 technology library and evaluated through OpenROAD. The task currently restricts the real optimization action to buffer insertion. This limited action scope was used to validate the complete execution path before extending the benchmark to additional transformations such as gate resizing, threshold-voltage swapping, cloning, or logic restructuring.

The current controller is a reasoning-based agent rather than a fully trained reinforcement-learning policy. It operates through the same RL-compatible environment interface and uses trial evaluation with state restoration to compare candidate actions. Accordingly, the experiments in this chapter are intended to validate the framework, interfaces, and real backend execution path. They do not establish state-of-the-art timing optimization performance.

Earlier stages of the project explored graph-based circuit representation, lightweight QoR prediction, and a standalone DQN-style OpenROAD prototype. These experiments helped identify requirements for state representation, backend integration, and modular action ex-

ecution. However, they are not treated as the primary experimental system in this chapter. The remainder of the chapter therefore focuses on the current RL-CHIP-Bench implementation and uses the earlier prototypes only where they help explain a specific design decision.

3.2 RL-CHIP-Bench Architecture

RL-CHIP-Bench is implemented as a layered software architecture that separates benchmark control, environment interaction, semantic action representation, backend execution, QoR evaluation, and agent decision making. The purpose of this separation is to prevent the optimization policy from becoming tightly coupled to one design, one backend script, or one local experiment configuration.

This architecture is motivated by the need to separate experimental control from agent-specific decision logic. Reproducible ML-for-EDA experiments require consistent task definitions and evaluation interfaces so that different optimization methods can be compared under the same conditions [2]. In RL-CHIP-Bench, this principle is implemented through separate benchmark, environment, semantic-action, backend, and agent layers.

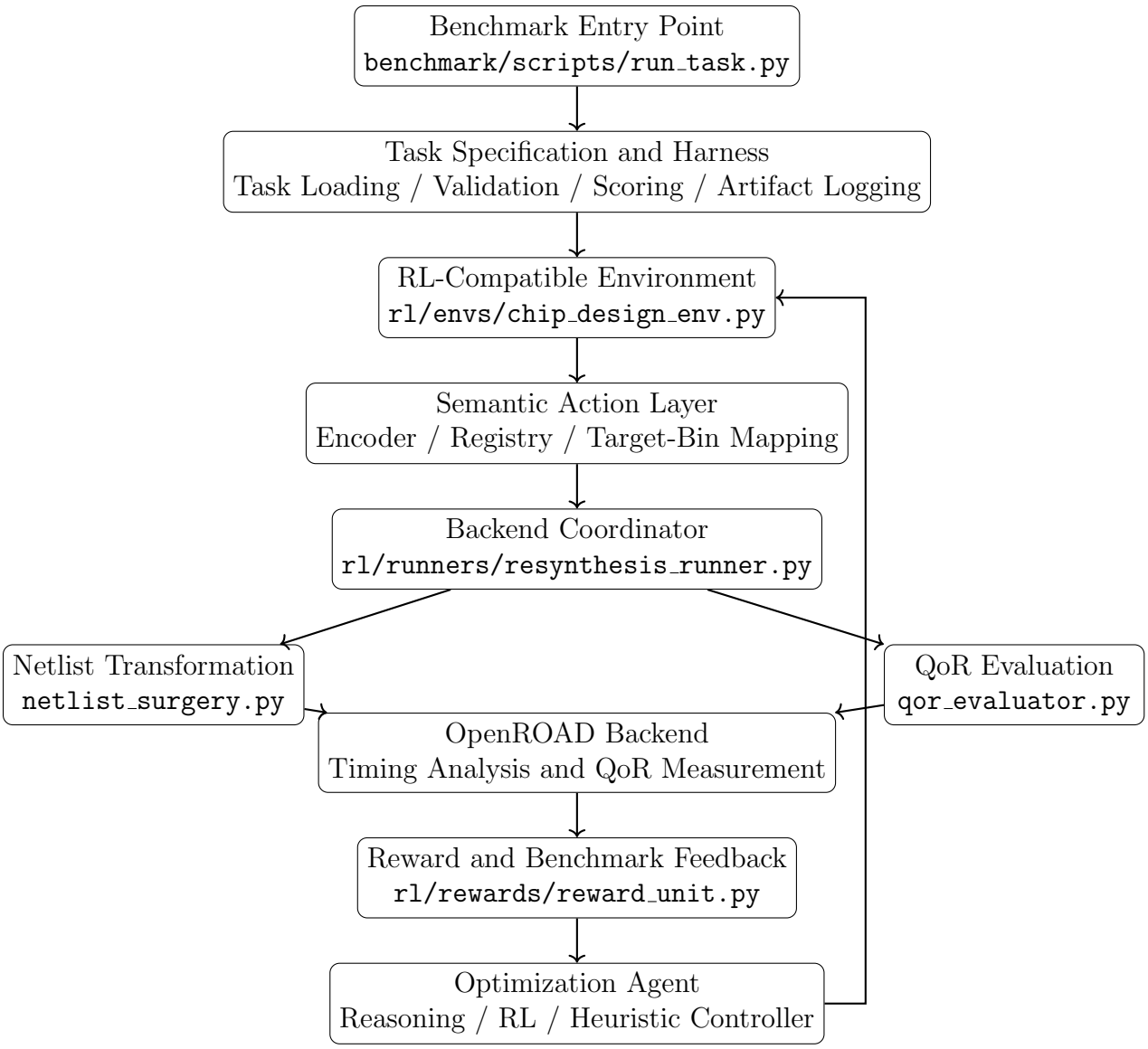

Figure 3.1: Implementation architecture of RL-CHIP-Bench. The benchmark harness constructs the task and runtime environment, the environment converts agent decisions into semantic actions, the backend runner applies design transformations and invokes OpenROAD, and the resulting QoR measurements are converted into benchmark feedback. The agent is separated from backend execution so that different decision-making methods can reuse the same infrastructure.

Figure 3.1 shows the implementation-oriented architecture of RL-CHIP-Bench. Unlike the conceptual overview presented in Chapter 1, this figure identifies the major software modules used by the current execution path.

Benchmark Control Layer

The execution process begins at the benchmark control layer. A benchmark run is launched through `benchmark/scripts/run_task.py`, which loads a task specification and constructs the runtime components required for the experiment.

Task information is represented through the benchmark harness, including modules for task loading, task specification, submission validation, scoring, and artifact logging. The task specification defines the design resources, backend configuration, episode constraints, reward settings, and the semantic actions permitted for that benchmark.

Keeping task configuration outside the agent implementation is important for reproducibility. The optimization method should not silently redefine the design, technology files, or evaluation objective. Instead, these properties are controlled by the benchmark and supplied to the runtime environment.

Environment Layer

The main environment implementation is `rl/envs/chip_design_env.py`. It provides the standardized sequential interaction interface used by both reinforcement-learning and

reasoning-based agents.

At the beginning of an episode, the environment initializes the runner and obtains the initial QoR state. During each optimization step, it receives an encoded action, checks whether the action is permitted by the current benchmark task, converts it into a semantic transformation, and sends it to the backend runner.

The environment itself does not directly modify Verilog or invoke OpenROAD. This separation is intentional: the environment controls the decision-making protocol, whereas the runner and backend layers control design execution.

Semantic Action Layer

The action layer converts a fixed-dimensional agent action into a design-dependent transformation. The current action representation has the form

$$a = [a_{\text{category}}, a_{\text{target}}, a_{\text{sub}}], \tag{3.1}$$

where a_{category} identifies the transformation family, a_{target} identifies a target bin, and a_{sub} specifies the operation within that family.

The corresponding implementation is organized through `rl/actions/encoder.py`, `rl/actions/registry.py`, and `rl/actions/target_bin.py`. The encoder converts the compact action into a semantic representation that can be understood by the backend runner. This allows the external action interface to remain fixed even though different designs contain

different instance names and numbers of candidate targets.

This action abstraction is one of the main architectural choices of the framework. A direct action space over every gate or net would change whenever the benchmark design changes. The semantic layer instead maintains a stable agent-facing representation while resolving design-specific targets internally.

Backend Execution Layer

The main backend coordinator in the current validated flow is `rl/runners/resynthesis_runner.py`. It receives the decoded semantic action from the environment and coordinates the corresponding design modification and QoR evaluation.

Concrete netlist transformations are handled through `rl/runners/netlist_surgery.py` and the associated transformation scripts. After an edit is applied, QoR evaluation is performed through `rl/runners/qor_evaluator.py`, which invokes the OpenROAD evaluation flow and returns the resulting timing and design-quality measurements.

This separation between semantic actions and physical execution is important because the same agent interface can remain unchanged even if the backend implementation evolves. For example, a future implementation could replace a text-based edit script with an OpenROAD API call while preserving the same high-level action representation.

QoR and Reward Feedback

The backend returns a QoR snapshot describing the modified design. The current environment tracks timing-related metrics such as worst negative slack (WNS) and total negative slack (TNS), together with additional design-quality signals when they are available.

The reward layer, implemented through `rl/rewards/reward_unit.py`, converts the measured before-and-after QoR change into a scalar feedback signal. The environment then returns the updated observation, reward, and episode status to the agent.

An important distinction is that the reward is not the original source of design information. The primary measurements are produced by OpenROAD. Reward computation is a transformation of those measurements for agent interaction, while the original QoR values are preserved for later evaluation.

Agent Layer

The agent layer is intentionally separated from the benchmark and backend infrastructure. An agent observes the state provided by the environment and determines which action to submit next, but it does not directly control OpenROAD or edit the design files.

The currently validated controller is implemented as a reasoning-based submission. Its detailed behavior is described later in this chapter. The same environment interface, however, is designed to support future reinforcement-learning, heuristic, or graph-based controllers without changing the backend execution path.

This separation creates a clear system boundary: the benchmark defines the task, the environment defines the interaction protocol, the backend determines the measured effect of an action, and the agent determines how actions are selected.

3.3 RL-Compatible Environment and Sequential Optimization Interface

RL-CHIP-Bench exposes chip optimization through a sequential agent–environment interface. The underlying formulation follows the reinforcement-learning interaction model described in Chapter 2: an agent observes the current state, selects an action, receives feedback from the environment, and continues until the episode ends [15]. The implementation follows a Gymnasium-style software interface using `reset()` and `step()`, while the environment hides the details of netlist modification and OpenROAD execution from the agent.

The main implementation of this layer is `rl/envs/chip_design_env.py`. The environment is responsible for coordinating observations, encoded actions, task-level action restrictions, reward computation, and episode state. It does not directly implement the physical design transformations. Instead, backend execution is delegated to the runner layer described later in this chapter.

This separation is important because the environment defines the optimization problem, whereas the backend defines how a selected transformation is physically executed. The same

agent-facing interface can therefore be maintained even when the backend implementation or benchmark task changes.

Agent–Environment Interface

At the beginning of an episode, `reset()` initializes the backend runner, retrieves the initial QoR measurements, rebuilds the target mapping for the current design, resets the internal step counter, and returns the initial observation.

During an optimization step, the agent supplies an encoded action to `step()`. The environment decodes the action, checks whether it is permitted by the current benchmark task, and forwards a valid semantic action to the backend runner. The runner applies the corresponding design transformation and returns updated QoR measurements. The environment then computes the reward and returns the next observation together with the episode status.

Conceptually, one environment interaction can be written as

$$(o_{t+1}, r_t, d_t, \mathcal{I}_t) = \mathcal{E}(o_t, a_t), \tag{3.2}$$

where o_t is the current observation, a_t is the submitted action, r_t is the reward, d_t represents the episode termination or truncation status, and $\mathcal{I}_t$ contains diagnostic information about the execution.

The use of a standardized interaction interface is consistent with the broader design of

reinforcement-learning benchmarks, where the task execution mechanism is separated from the internal learning algorithm [7]. In RL-CHIP-Bench, this separation is particularly important because one environment transition may involve a complete EDA backend evaluation rather than an inexpensive simulator update.

Observation Design

The current RL-CHIP-Bench implementation uses a compact QoR-level observation rather than exposing the complete circuit graph directly to the agent. In the validated environment, the observation vector is defined as

$$o_t = [\mathrm{WNS}_t, \mathrm{TNS}_t, P_t, W_t, \rho_t], \tag{3.3}$$

where WNS_t and TNS_t describe timing quality, P_t denotes power when available, W_t denotes wirelength when available, and ρ_t represents the normalized progress through the current episode.

The final component is calculated from the current environment step and the maximum episode length:

$$\rho_t = \frac{t}{T_{\max}}. \tag{3.4}$$

The timing components are particularly important because WNS and TNS directly indicate whether timing violations are improving or degrading. These metrics are also used in

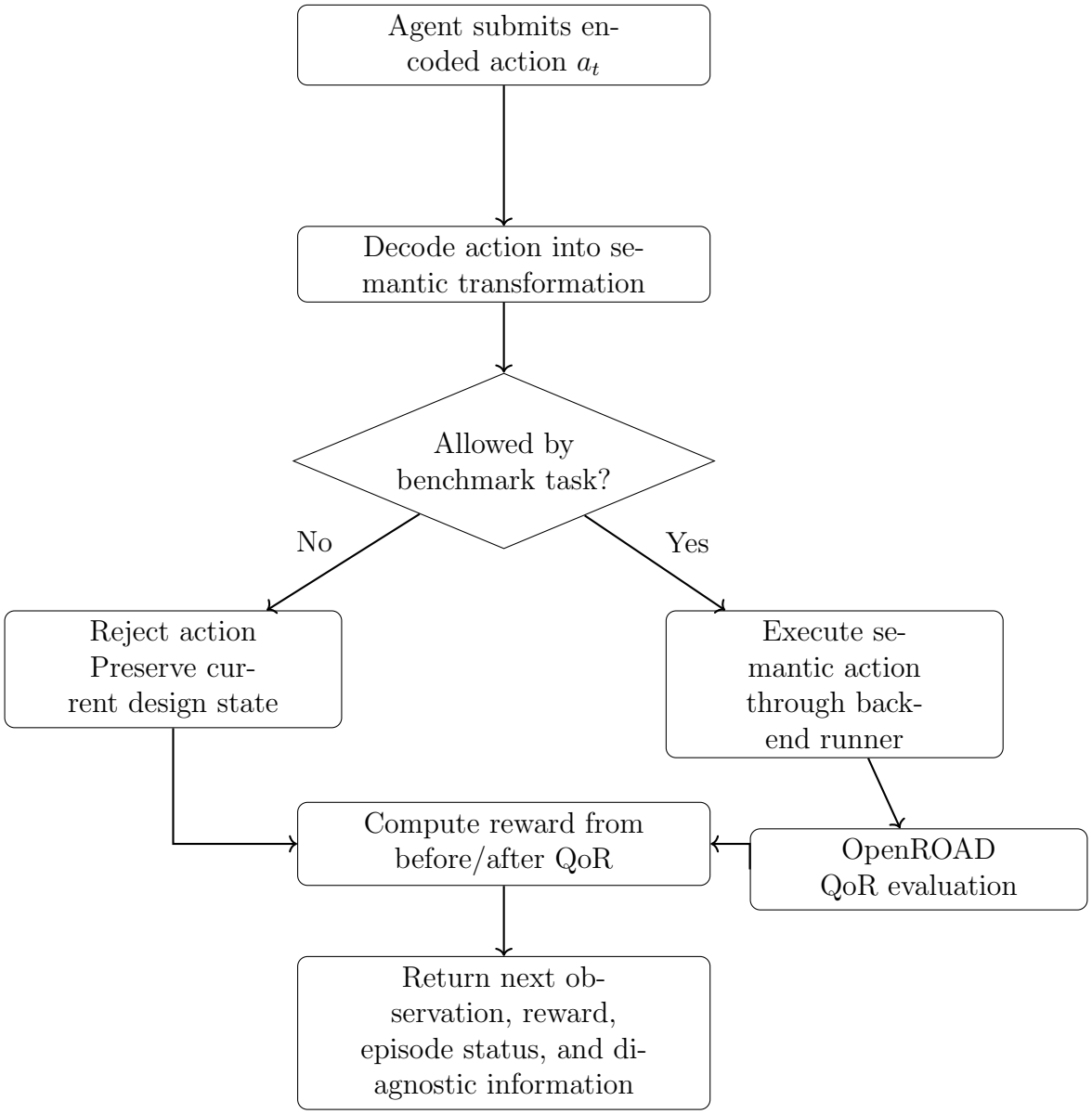

Figure 3.2: Execution flow of one RL-CHIP-Bench environment step. A submitted action is decoded and checked against task-specific restrictions before backend execution. Valid actions are evaluated using the real EDA backend, while invalid or disallowed actions are rejected without modifying the design. The resulting QoR measurements are converted into the next observation and reward.

learning-based timing optimization work such as RL-Sizer and IR-aware ECO optimization [9, 8].

The compact observation was selected deliberately for the current validated implementation. A full gate-level representation may contain thousands or more circuit objects and would require additional graph extraction, feature normalization, and agent-side processing. Using a small QoR vector first made it possible to validate the interaction protocol, backend integration, reward computation, and benchmark execution independently of a more complex state encoder.

This compact representation also has an important limitation. Global QoR values describe the overall design condition but provide little information about where a timing problem is located. Two different circuits may have similar WNS and TNS values while requiring very different local actions. Graph embeddings, critical-path features, and instance-level timing features are therefore natural future extensions, as discussed in Chapter 2.

Action Interface

The environment exposes a fixed-dimensional discrete action vector

$$a_t = [c_t, b_t, u_t], \tag{3.5}$$

whose three integer components select an action category, a target bin, and a category-dependent sub-action, respectively. The precise domains and target-mapping mechanism are

defined in Section 3.4.

This abstraction addresses a practical difficulty in chip optimization: different designs contain different numbers of gates, nets, and candidate targets. A direct action space over all physical objects would therefore change with the benchmark design. Large and design-dependent action spaces are also a known challenge in learning-based chip optimization [11, 9].

The agent-facing action representation remains fixed, while the environment internally maps the target component to an actual circuit object and converts the compact vector into a semantic transformation. The details of this mapping and the associated target-bin design are presented in Section 3.4.

Task-Level Action Restrictions

The framework distinguishes between the global action space supported by the environment and the subset of actions permitted by a particular benchmark task. This distinction allows the environment interface to remain unchanged while individual tasks restrict the optimization problem.

After decoding an action, the environment compares the semantic operation against the allowed-action specification supplied by the benchmark task. If the operation is not permitted, the backend is not modified. The action is marked as unsuccessful and diagnostic information is returned to the agent.

This approach avoids rebuilding the action-space dimensions for every benchmark task. It also makes task restrictions explicit and reproducible. For example, the current validated benchmark task permits only buffer insertion even though the broader framework contains representations for additional optimization categories.

The environment also exposes high-level action availability information that can be used by an agent to avoid obviously disallowed categories. However, detailed validity may still depend on the specific target and operation, so action legality cannot be determined from the top-level category alone.

Reward Computation

The reward formulation used in the current RL-CHIP-Bench environment is implemented in `rl/rewards/reward_unit.py`. Unlike the generic multi-objective reward discussion in Chapter 2, this logic is specific to the framework developed in this thesis. It compares the backend-derived QoR measurements before and after an environment action and converts the measured change into a scalar feedback signal.

The reward logic is conditional on the timing state before the action. At a high level, it can be represented as

$$r_t = \begin{cases} R_{\text{repair}}\left(\mathbf{q}_t, \mathbf{q}_{t+1}\right), & \text{WNS}_t < 0, \\ R_{\text{feasible}}\left(\mathbf{q}_t, \mathbf{q}_{t+1}\right), & \text{WNS}_t \geq 0, \end{cases} \tag{3.6}$$

where $\mathbf{q}_t$ and $\mathbf{q}_{t+1}$ denote the backend-derived QoR measurements before and after the action. R_{repair} prioritizes recovery from timing violations, whereas R_{feasible} allows secondary QoR improvement while penalizing actions that reintroduce a timing violation.

When the previous design still violates timing, indicated by negative WNS, the reward prioritizes timing repair. An improvement in WNS contributes positively to the reward, while degradation receives a penalty. Improvements in secondary metrics such as power or wirelength can contribute additional smaller terms when those measurements are available.

When the previous design already satisfies timing, the reward shifts emphasis toward secondary QoR improvement while penalizing actions that reintroduce a timing violation. This conditional behavior reflects the practical priority that timing feasibility should not normally be sacrificed for secondary optimization objectives.

The important implementation principle is that reward is computed from backend-derived measurements. The environment does not assign an action a positive reward merely because the transformation is syntactically valid. A circuit edit must first be executed and evaluated before its measured effect is known.

The reward implementation also supports penalties for unsuccessful or invalid actions and applies configurable clipping to keep the scalar feedback within a bounded numerical range. The benchmark task can therefore adjust reward configuration without changing the environment interface.

Episode Control

The current environment uses a benchmark-defined maximum step count to limit the duration of an optimization episode. The environment therefore distinguishes between task termination and truncation caused by the step budget.

In the current implementation, reaching the maximum number of steps produces a truncated episode rather than asserting that a final timing-closure condition has been reached. This choice is important because the current benchmark does not assume that every episode will successfully close timing.

A fixed episode budget also provides a common resource constraint across agents. Without such a limit, an optimization strategy that performs substantially more backend evaluations could not be compared fairly with one that uses fewer interactions.

State Save and Restore

In addition to the standard sequential interface, the environment supports state save-and-restore operations for planning-style controllers. A saved state includes the environment information required to return to the current optimization point together with the corresponding runner state.

This capability enables an agent to temporarily test a candidate transformation, observe its backend-derived QoR result, restore the original design state, and then evaluate another candidate. The mechanism is especially important for the reasoning-based agent described

later in this chapter.

State restoration is not required by a conventional online reinforcement-learning policy, which normally commits each action as part of its trajectory. It is nevertheless useful in the current framework because the same environment is designed to support both sequential learning and local planning.

Algorithm 1 RL-CHIP-Bench Environment Step

Require: Encoded action a_t, current QoR q_t, task specification $\mathcal{T}$
Ensure: Next observation o_{t+1}, reward r_t, episode status, diagnostic information

$a_t^{\text{sem}} \leftarrow \textsc{DecodeAction}(a_t)$
if $\textsc{AllowedByTask}(a_t^{\text{sem}}, \mathcal{T})$ **then**
    $\text{success}, q_{t+1} \leftarrow \textsc{BackendApplyAndEvaluate}(a_t^{\text{sem}})$
else
    $\text{success} \leftarrow \text{false}$
    $q_{t+1} \leftarrow q_t$
end if
$r_t \leftarrow \textsc{ComputeReward}(q_t, q_{t+1}, \text{success})$
$o_{t+1} \leftarrow \textsc{BuildObservation}(q_{t+1}, t+1)$
$\text{truncated} \leftarrow (t+1 \geq T_{\max})$
$\text{info} \leftarrow \textsc{BuildDiagnostics}(a_t^{\text{sem}}, q_{t+1}, \text{success})$
return $o_{t+1}, r_t, \text{terminated}, \text{truncated}, \text{info}$

Algorithm 1 summarizes the main control flow of one environment transition. The key design feature is that action decoding and task validation occur before backend execution. A valid action triggers a real design transformation and QoR evaluation, whereas a disallowed action preserves the current design state. In both cases, the environment returns the same standardized interaction structure to the agent.

3.4 Semantic Action Representation and Target Mapping

A central difficulty in applying reinforcement learning to chip optimization is that the set of physical design objects changes from one circuit to another. One design may contain a few thousand candidate instances, while another may contain substantially more. If the agent action directly referenced instance names or net identifiers, the action space would change with every benchmark design.

RL-CHIP-Bench addresses this problem by separating the agent-facing action representation from the design-dependent physical target. The environment exposes a fixed-dimensional action vector, while the action layer resolves that vector into a semantic operation and a concrete circuit object before backend execution.

This abstraction follows the broader observation that learning-based chip optimization often involves large and design-dependent action spaces [11, 9]. The purpose of the current encoding is not to eliminate this complexity, but to provide a stable interface through which different agents can interact with different benchmark designs.

Fixed-Dimensional Action Encoding

The current action representation is a three-dimensional vector of discrete integer indices,

$$a_t = [c_t, b_t, u_t] \in \mathbb{Z}^3_{\geq 0}, \tag{3.7}$$

where

$$c_t \in \{0, \ldots, C-1\}, \qquad b_t \in \{0, \ldots, B-1\}, \qquad u_t \in \{0, \ldots, U-1\}. \tag{3.8}$$

Here, c_t is the action-category index, b_t is the target-bin index, and u_t is the category-dependent sub-action index. The quantities C, B, and U denote the configured sizes of the corresponding discrete dimensions. In the current implementation, the Gymnasium action space is represented as a `MultiDiscrete` space rather than as one-hot vectors or continuous values.

The action-category index c_t selects one of the semantic transformation families registered by the action layer. The broader framework contains representations for operations such as buffer insertion or deletion, gate resizing, threshold-voltage swapping, and logic restructuring. The benchmark task may subsequently restrict which of these semantic operations are executable in a particular experiment. The target-bin component selects a design-dependent target through a fixed-size index space, while the sub-action specifies the operation variant within the selected category.

The implementation is organized through `rl/actions/encoder.py` and `rl/actions/registry.py`. The registry associates compact integer identifiers with higher-level operation meanings, while the encoder converts the numerical action into a structured semantic representation.

The important design principle is that the agent does not directly select a backend script.

Instead, it selects an abstract operation. Backend-specific execution is resolved later by the runner layer.

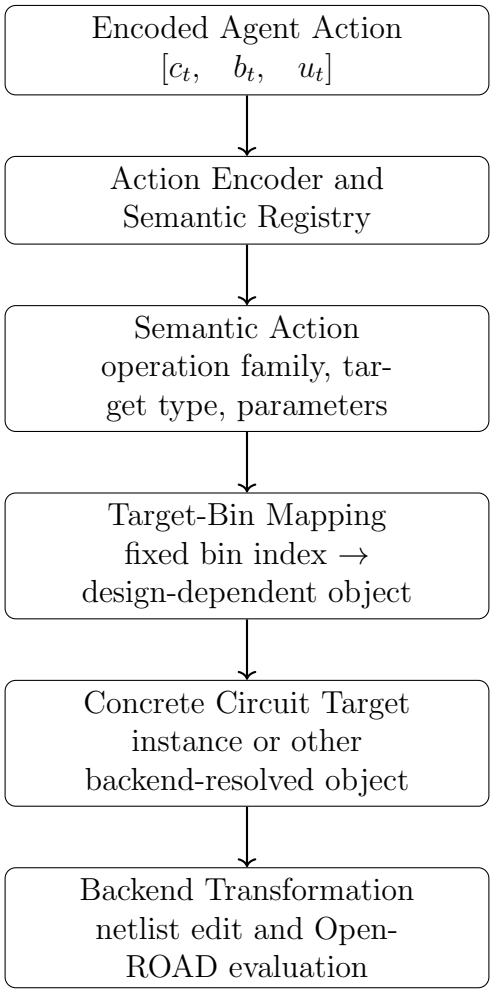

Figure 3.3: Mapping from a fixed-dimensional discrete agent action to a concrete design transformation. The action components are integer indices whose configured ranges define the agent-facing action space. The target-bin index is then resolved to a design-dependent circuit object before backend execution.

Semantic Action Representation

After decoding, the numerical action is represented internally as a semantic action rather than as a raw integer tuple. This representation separates the operation family, operation name, target information, and operation-specific parameters.

For example, two actions may use the same target bin while requesting different transformations. Conversely, the same semantic transformation can be applied to different design objects without changing the agent-facing action format.

This intermediate representation is useful for three reasons. First, it separates learning logic from backend script names. Second, it allows benchmark tasks to restrict actions using meaningful operation categories rather than low-level numerical identifiers. Third, it makes the execution path easier to extend because new backend implementations can consume the same semantic action representation.

The current code maintains compatibility with older runner interfaces, but these backward-compatibility fields are implementation details rather than part of the conceptual action model. The thesis therefore focuses on the semantic abstraction rather than listing the internal mapping keys used by the Python implementation.

Target Binning

The target component uses a configurable fixed-size index space. Let B denote the number of target bins configured for a particular environment or benchmark setting. Although the

number of available circuit instances may vary between designs, B is kept constant within that setting so that the agent-facing action-space shape does not change across episodes or designs evaluated under the same configuration.

RL-CHIP-Bench implements this abstraction through `rl/actions/target_bin.py`.

The mapper assigns design instances to a fixed number of bins. During environment initialization, the current instance list is used to construct the mapping between bin indices and concrete design objects. The agent then selects a bin rather than selecting an instance name directly.

Let the set of design instances be

$$\mathcal{V} = \{v_1, v_2, \ldots, v_N\}, \tag{3.9}$$

and let the environment expose B target bins. The mapping can be written abstractly as

$$\phi : \mathcal{V} \to \{0, 1, \ldots, B-1\}. \tag{3.10}$$

The value of B is therefore an environment configuration parameter rather than a property of the circuit itself. Different benchmark configurations may choose different values of B, but changing B also changes the second dimension of the agent action space and therefore changes the policy interface.

The current implementation uses a deterministic hash-based assignment so that the mapping can be reconstructed consistently from the same design instance names. For a fixed value of B, this provides a simple way to maintain the same target-index dimension even when the number and identities of circuit instances change across designs.

The main benefit of this design is interface stability. A reinforcement-learning policy can always output an action with the same dimensional structure even when the benchmark design changes. The environment resolves the selected bin to the corresponding physical object internally.

The choice of B therefore introduces an additional design trade-off. A smaller value reduces the action-space size but increases the likelihood of collisions, whereas a larger value provides finer target resolution at the cost of a larger discrete action dimension. The current thesis does not present an optimization study over B; it is treated as a configurable environment parameter.

Empty Bins and Mapping Collisions

Fixed-size hashing introduces two practical limitations. First, some target bins may contain no valid design object. Second, multiple instances may map to the same bin.

An empty bin does not correspond to a useful physical target and therefore cannot directly produce a meaningful transformation. A collision creates ambiguity because one fixed agent action may correspond to more than one candidate object.

These effects illustrate a trade-off between a stable RL-compatible interface and exact control over the physical design. A fully explicit target space would provide precise object selection but would vary with every design. A fixed hashed space provides a stable interface but introduces approximate target resolution.

The current implementation accepts this trade-off because the primary goal is to validate the benchmark and environment architecture. However, target quality directly affects optimization quality. Future versions should therefore consider timing-aware candidate filtering, critical-path-based target generation, or more informative action masks rather than relying only on static hashing.

Interaction with Benchmark Task Restrictions

The semantic action layer also provides the connection between the global environment action space and task-specific benchmark constraints.

A benchmark task may expose only a subset of the transformation families supported by the repository. The environment first decodes the numerical action into its semantic form and then checks whether that semantic operation is permitted by the current task specification. This allows task restrictions to be expressed in terms of meaningful operations rather than numerical action IDs.

The current validated benchmark task restricts execution to buffer insertion. Other semantic categories are represented in the broader framework but are not treated as validated

actions for the current experiment. This distinction is important because the existence of a transformation script does not imply that the corresponding operation has been fully tested under the current benchmark configuration.

The action abstraction therefore serves two purposes: it provides a stable interface for agents and provides a semantic boundary at which benchmark constraints can be enforced.

Design Implications

The current action representation is intentionally simple, but it exposes an important system-design trade-off. Reinforcement-learning algorithms benefit from a fixed and compact action space, while EDA optimization naturally operates over irregular and design-dependent objects.

The semantic action layer resolves this mismatch by introducing an intermediate representation between the agent and the physical design. This improves modularity and allows different controllers to share the same backend interface. At the same time, the quality of target mapping becomes part of the optimization problem itself.

For the current benchmark, this abstraction is sufficient to validate the complete agent-to-backend execution path. For stronger optimization agents, however, a more informative target representation will likely be required. Possible extensions include critical-path-aware candidate sets, graph-based target embeddings, and hierarchical action selection in which the agent first selects an optimization region and then selects a concrete operation within

that region.

3.5 Benchmark Harness and Task Specification

The benchmark harness is the control layer that turns a chip-design optimization problem into a reproducible experimental task. Its role is to define the design resources, execution constraints, allowed semantic actions, runtime configuration, artifact logging, and scoring procedure independently of the optimization agent.

This separation is important for reproducibility. In ML-driven EDA, experimental results can depend strongly on design files, technology libraries, timing constraints, tool versions, and evaluation settings. A benchmark framework should therefore control these factors explicitly rather than allowing them to remain distributed across local scripts or agent-specific configurations [2]. Similar reproducibility concerns also motivate tracked task definitions and consistent reporting in general reinforcement-learning benchmarking [6].

In the implementation developed in this thesis, these concerns are addressed concretely through task manifests and benchmark-controlled runtime construction: the design resources, technology configuration, action restrictions, episode settings, and evaluation procedure are loaded from the task specification rather than selected independently by the agent.

In RL-CHIP-Bench, the benchmark harness loads a task specification, validates required resources and submission compatibility, constructs the runtime environment and backend, executes the selected agent, records the resulting interaction sequence, and writes structured

outputs for later evaluation.

Benchmark Task Model

A benchmark task represents one complete optimization problem. It specifies the circuit being optimized, the technology configuration, the backend evaluator, the episode constraints, the reward configuration, and the subset of semantic actions that the agent is permitted to use.

Conceptually, a task can be represented as

$$\mathcal{T} = (D, K, B, A_{\text{allow}}, R, H, M)\,, \tag{3.11}$$

where D denotes the design resources, K denotes the technology configuration, B denotes the backend configuration, A_{allow} is the set of allowed semantic actions, R is the reward configuration, H is the episode horizon, and M denotes the metrics used for evaluation and scoring.

The purpose of this abstraction is to make the optimization problem independent of the agent implementation. Two agents evaluated on the same task should receive the same design, constraints, action restrictions, backend behavior, and scoring rules. Their difference should come from the decision-making method rather than from hidden changes to the experimental setup.

Task Manifest

Each benchmark task is described through a structured manifest. In the current repository, task definitions are stored under

`benchmark/tasks/`.

The manifest is loaded by the benchmark harness and converted into a structured task representation before runtime construction. It provides the information required to resolve design paths, technology resources, backend settings, episode limits, reward parameters, and allowed actions.

Using an explicit task manifest reduces hard-coded dependencies. In an earlier standalone prototype, configuration values could be distributed across scripts, local paths, and environment-specific settings. In the benchmark-oriented implementation, the task manifest provides a single control point for the experimental configuration.

This also makes task differences easier to inspect. For example, one benchmark may allow only buffer insertion, while another may later expose gate resizing or multiple semantic action categories without requiring a new agent interface.

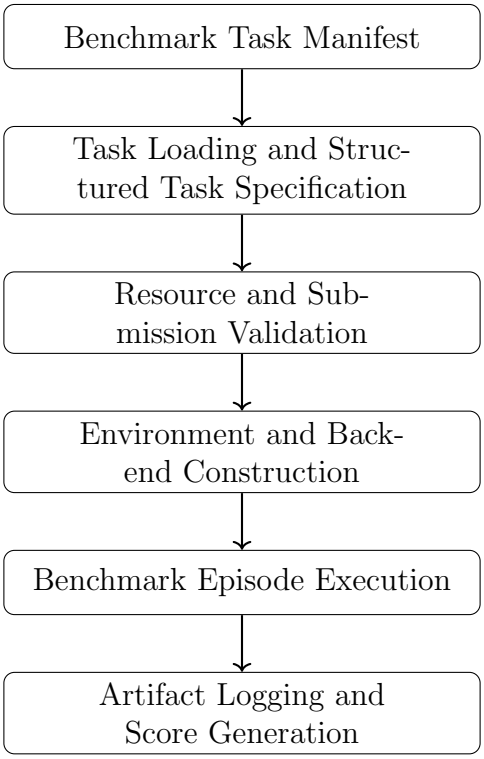

Figure 3.4: Benchmark control flow in RL-CHIP-Bench. A task manifest is converted into a structured runtime specification, validated before execution, and used to construct the environment and backend. The harness then executes the selected submission and records structured artifacts and benchmark outputs.

Resource and Submission Validation

Before an optimization episode begins, the harness validates the task resources and the selected submission. This step is necessary because a real EDA-backed experiment depends on multiple external resources that must remain mutually consistent.

The required resources may include a gate-level Verilog netlist, timing constraints, technology-library information, LEF and Liberty data, backend configuration, and OpenROAD execu-

tion scripts. Missing or incorrectly resolved resources can cause failure deep inside the backend flow, where the resulting error is harder to diagnose.

The benchmark therefore performs validation before runtime execution. At a high level, the validation stage checks whether the task specification can be resolved into the required design and technology resources and whether the selected submission is compatible with the task definition.

Submission validation is also connected to action restrictions. A benchmark task can define which semantic transformations are legal for that experiment. The harness and environment use this specification to prevent a submission from silently executing operations outside the intended task scope.

This design moves many configuration errors from backend runtime into an earlier and more explicit validation stage, which improves both debugging and reproducibility.

End-to-End Benchmark Execution

A benchmark run coordinates task construction, environment initialization, agent execution, backend evaluation, and artifact generation. Algorithm 2 summarizes the main control flow.

Algorithm 2 End-to-End RL-CHIP-Bench Execution

Require: Task identifier τ, submission identifier σ
Ensure: Benchmark artifacts and final score

$\mathcal{T} \leftarrow \textsc{LoadTask}(\tau)$
$\textsc{ValidateTaskResources}(\mathcal{T})$
$\textsc{ValidateSubmission}(\sigma, \mathcal{T})$
$\mathcal{E} \leftarrow \textsc{ConstructEnvironment}(\mathcal{T})$
$\mathcal{A} \leftarrow \textsc{ConstructAgent}(\sigma, \mathcal{T})$
$(o_0, \text{info}_0) \leftarrow \textsc{Reset}(\mathcal{E})$
$t \leftarrow 0$
while not terminated **and not** truncated **do**
    $a_t \leftarrow \textsc{SelectAction}(\mathcal{A}, o_t, \text{info}_t)$
    $(o_{t+1}, r_t, \text{terminated}, \text{truncated}, \text{info}_{t+1}) \leftarrow \textsc{Step}(\mathcal{E}, a_t)$
    $\textsc{LogStep}(t, a_t, r_t, \text{info}_{t+1})$
    $t \leftarrow t + 1$
end while
$m_{\text{final}} \leftarrow \textsc{CollectFinalMetrics}(\mathcal{E})$
$s \leftarrow \textsc{ComputeBenchmarkScore}(m_{\text{final}}, \mathcal{T})$
$\textsc{WriteArtifacts}(\mathcal{T}, \sigma, m_{\text{final}}, s)$
return artifacts and score

Algorithm 2 shows that the benchmark harness controls the experiment outside the agent itself. The agent is responsible only for selecting actions from the information exposed by the environment. Task loading, resource validation, runtime construction, step logging, final metric collection, and scoring remain benchmark responsibilities.

This separation is important for later agent comparison. A reinforcement-learning agent and a reasoning-based agent may use very different internal decision logic, but they can still be evaluated under the same task and execution protocol.

Artifact Logging and Scoring

The benchmark harness records structured outputs rather than relying only on terminal messages or screenshots. This is important because console logs are difficult to compare across runs and may omit the context required to reconstruct an experiment.

The current framework records information such as the task identity, submission identity, execution status, action history, backend results, final QoR metrics, and benchmark-level summary information. These outputs provide a machine-readable record of the experiment and support later comparison between submissions.

Two artifacts used in the current validated flow are

`run_summary.json`

and

`final_metrics.json`.

The first records run-level summary information, while the second records the final QoR measurements used for analysis. These files should be interpreted as experimental records rather than as research results by themselves. Their importance is that they preserve the measured output in a form that can be inspected, compared, or processed automatically.

Structured experiment tracking is consistent with reproducibility goals in both reinforcement-learning benchmarking and ML-for-EDA research [6, 2].

Current Validated Task

The current real experiment in this thesis uses the benchmark identifier `TASK_0013`. This identifier refers to the task manifest used to define the validated RL-CHIP-Bench execution setup; it is not itself an optimization algorithm.

The task uses an AES gate-level design with the Nangate45 technology setup and a real OpenROAD backend. Its purpose is to validate the complete path from task loading and agent interaction to semantic design modification, backend QoR evaluation, reward feedback, and structured artifact generation.

The current task intentionally exposes a narrow action scope. Although RL-CHIP-Bench contains semantic representations and scripts for several transformation families, the validated task permits buffer insertion only. Internally, this action is identified as

`buffer_insert_delete.insert`.

This internal identifier represents the insertion sub-operation of the buffer action family. Restricting the task to one validated transformation reduces ambiguity during system integration and allows the framework behavior to be tested before broader action combinations are introduced.

Table 3.1: Configuration of the current validated RL-CHIP-Bench task.

Component	**Current validated configuration**
Benchmark identifier	`TASK_0013`
Design	AES gate-level design
Technology	Nangate45
Backend	OpenROAD
Agent type	Reasoning-based submission
Allowed semantic operation	Buffer insertion
Internal action identifier	`buffer_insert_delete.insert`
Evaluation basis	Backend-derived QoR measurements
Output format	Structured benchmark artifacts and final metrics

The validated task should therefore be interpreted as an integration benchmark rather than as evidence that the complete transformation library has already been optimized or tested across multiple designs. Its main role in the current thesis is to demonstrate that the benchmark harness, environment, semantic action layer, real backend, agent interface, and artifact pipeline can operate together under one controlled task specification.

3.6 Backend Execution and OpenROAD-Based QoR Evaluation

The backend execution layer converts a semantic optimization action into a concrete design modification and obtains the corresponding QoR measurements. In the current validated implementation, this process includes netlist modification, OpenROAD execution, timing analysis, and report parsing. The measured result is then returned to the environment for

reward computation and benchmark analysis.

In the current RL-CHIP-Bench implementation, OpenROAD provides the primary tool-backed evaluation path. OpenROAD is an open-source digital implementation framework that supports timing analysis and physical-design optimization and is suitable for reproducible ML-for-EDA experimentation [3]. The current framework uses OpenROAD to evaluate timing-related QoR after each accepted design transformation.

The backend is intentionally separated from the environment and agent layers. The environment determines when an action should be executed, while the backend determines how that action modifies the design and what measured QoR results from the modification. This separation allows the optimization interface to remain stable even if the internal execution mechanism changes.

Backend Coordination

The main backend coordinator in the current validated flow is `rl/runners/resynthesis_runner.py`. The runner receives a decoded semantic action from the environment, applies the corresponding design transformation, invokes QoR evaluation, and returns the resulting metrics.

From the environment perspective, the runner provides a compact execution interface. The environment does not need to know which script modifies the netlist or how OpenROAD reports are generated. It submits a semantic action and receives an updated QoR state.

From the backend perspective, the runner coordinates several lower-level responsibilities: design-state initialization, netlist transformation, OpenROAD invocation, report parsing, and state save or restore when required by planning-style agents.

This modularity also allows the same environment to interact with different runner implementations. A lightweight or mock backend can be useful for interface debugging, while the validated experiment uses the real OpenROAD-backed path.

Netlist Transformation

Concrete circuit edits are handled through the netlist transformation layer. The current execution path uses `rl/runners/netlist_surgery.py` together with transformation-specific scripts.

A high-level action such as buffer insertion must ultimately be translated into a legal modification of the gate-level design. This requires more than selecting an operation name. The backend must identify a valid target, instantiate a legal standard-cell object, preserve correct signal connectivity, and produce a modified design that can still be parsed by the EDA flow.

For the current validated task, the real transformation exercised by the benchmark is buffer insertion. The corresponding script used in the current flow is

`scripts/openroad/buffer_insert.py`.

Other transformation scripts exist in the broader repository, including support for buffer

deletion, gate resizing, threshold-voltage swapping, cloning, and logic restructuring. However, these operations are treated as extensible framework capabilities rather than as fully validated actions in the current benchmark experiment.

This distinction is important because the existence of a transformation script does not by itself establish that the action has been validated across tasks, technologies, or target conditions.

OpenROAD QoR Evaluation

After the working design has been modified, the backend invokes OpenROAD to evaluate the new design state. The current evaluation flow uses

`scripts/openroad/run_openroad_qor.tcl`.

This script provides the tool-level entry point for loading the design and technology resources, applying the timing constraints, running timing analysis, and producing the QoR information required by the Python runtime.

The current evaluation focuses primarily on timing-related measurements, especially worst negative slack (WNS) and total negative slack (TNS). These metrics were introduced in Chapter 2 and are commonly used for timing optimization and timing-repair evaluation [9, 8].

For an optimization step, the backend produces a before-and-after QoR pair

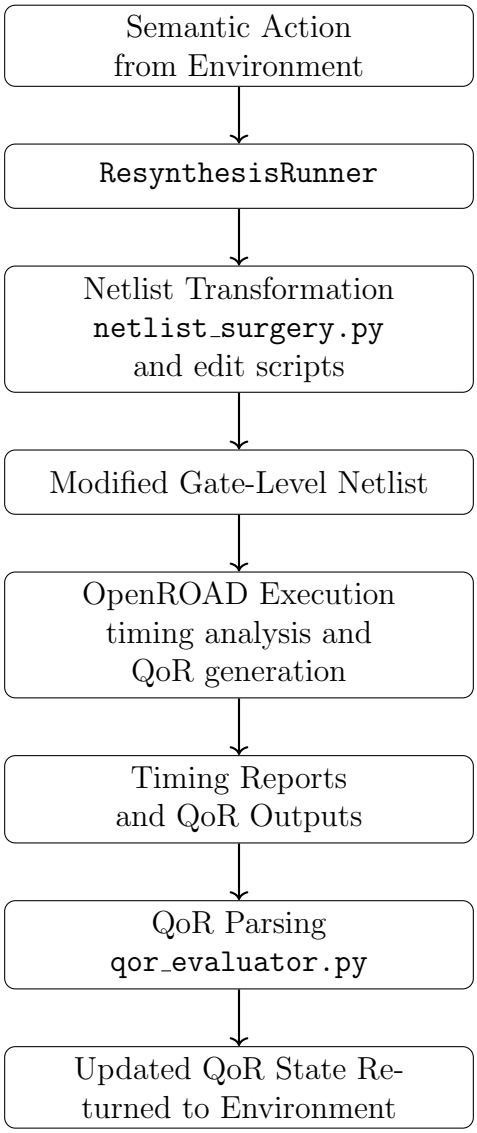

Figure 3.5: Backend execution path in the current RL-CHIP-Bench implementation. A semantic action is converted into a concrete netlist modification, evaluated through OpenROAD, and parsed into updated QoR measurements that are returned to the environment.

$$\mathbf{q}_t \rightarrow \mathbf{q}_{t+1}, \tag{3.12}$$

where each QoR snapshot contains the measurements available from the current backend configuration. The environment then uses the change between these states for reward computation and later benchmark analysis.

The key point is that the QoR values originate from the EDA backend rather than from a manually defined transition function. This gives the environment a more realistic connection to the design flow, although it also increases the cost of each interaction.

QoR Parsing and Normalization

Backend output is converted into a structured form by `rl/runners/qor_evaluator.py`. This layer isolates report parsing from the rest of the environment.

Separating parsing logic is important because EDA tools typically produce multiple forms of output, including reports, logs, and machine-readable files. The environment should not depend directly on the formatting of those files. Instead, the QoR evaluator converts the backend result into a consistent dictionary-like representation that can be used by the runner, reward unit, and artifact logger.

The parsed metrics are then used in two different ways. First, they provide the observation and reward inputs needed during agent interaction. Second, the original measured values are preserved for experiment analysis and benchmark output.

This distinction prevents the scalar reward from becoming the only record of design quality. A reward may combine several metrics, whereas the preserved QoR values allow later inspection of the actual timing or secondary metric changes.

A lightweight mock runner was useful during software debugging, but the experimental validation reported in this thesis uses the OpenROAD-backed execution path. Mock execution is therefore treated only as a development aid and is not used as evidence for the reported QoR results.

Managing Expensive Backend Evaluations

The high interaction cost discussed in Chapter 2 is not eliminated by the current RL-CHIP-Bench implementation. Each candidate that reaches the real backend may still require netlist modification, OpenROAD execution, timing analysis, report generation, and QoR parsing. The current system therefore does not claim to solve the general sample-efficiency problem of reinforcement learning for EDA.

Instead, the implementation uses several practical mechanisms to avoid some unnecessary backend evaluations during the current validation workflow.

First, the benchmark task explicitly restricts the executable transformation set. Although the framework contains semantic representations for several operation families, the current validated task permits only buffer insertion. This restriction reduces the executable search space during system validation and prevents the controller from issuing transformation types

that have not been validated under the active task.

Second, the reasoning-based controller generates a limited set of candidate semantic actions rather than exhaustively evaluating the complete encoded action space. Only these selected candidates are passed through trial evaluation. This reduces the number of actions considered for real-backend evaluation at each decision point, although the number of required OpenROAD calls still depends on the number of generated candidates.

Third, state save-and-restore allows the selected candidates to be evaluated from the same design state. This mechanism does not reduce the cost of an individual OpenROAD evaluation, but it avoids accumulating temporary transformations and makes the limited set of candidate evaluations directly comparable.

The current strategy can therefore be summarized as

large potential action space $\rightarrow$ task-level restriction $\rightarrow$ limited candidate generation $\rightarrow$ selected real-backend evaluations.

These mechanisms mitigate unnecessary backend execution in the current reasoning-based workflow, but they do not make EDA interaction inexpensive. In particular, each candidate that is actually tested may still require a full backend evaluation. As the candidate set grows, the execution cost grows with it.

Consequently, broader RL training will require stronger mechanisms for reducing the number of expensive evaluations. Prior work provides several examples, including local timing approximation in RL-Sizer and problem-specific action-space reduction in IR-aware

ECO optimization [9, 8]. Timing-aware target filtering, graph-based candidate ranking, reuse of previously collected transitions, and surrogate QoR models are additional directions for future versions of RL-CHIP-Bench rather than capabilities demonstrated by the current implementation.

Table 3.2: Main components of the current validated backend execution path.

Component	**Role in the current flow**
`ResynthesisRunner`	Coordinates semantic action execution and backend evaluation.
`netlist_surgery.py`	Provides the netlist-transformation interface used by the runner.
`buffer_insert.py`	Applies the buffer-insertion transformation used by the validated task.
`run_openroad_qor.tcl`	Runs OpenROAD timing and QoR evaluation.
`qor_evaluator.py`	Parses backend outputs into structured QoR measurements.
OpenROAD	Provides the real EDA timing-analysis backend.

The backend path establishes a complete connection between semantic agent actions and real EDA measurements. This capability addresses backend fidelity and system integration rather than evaluation cost: optimization decisions are translated into concrete design modifications and measured through the OpenROAD flow, while the interaction cost of those measurements remains a limitation discussed above.

The next section describes how the current reasoning-based agent uses this same environment and backend infrastructure to evaluate candidate actions through rollback-based trial execution.

3.7 Reasoning-Based Agent and Candidate Evaluation

The current validated RL-CHIP-Bench submission uses a reasoning-based controller rather than a fully trained reinforcement-learning policy. In the repository, this submission is named `arc_reasoning`. Its role is to demonstrate that the benchmark, environment, semantic action layer, and real OpenROAD backend can support a planning-style decision process through the same interface intended for future RL agents.

The reasoning controller does not modify the design directly. Instead, it observes the information exposed by the environment, proposes candidate semantic transformations, evaluates selected candidates through temporary environment interactions, and submits the selected action through the standard benchmark interface.

The implementation is primarily organized through

`submissions/arc_reasoning/agent.py`

and

`rl/reasoning/arc_core.py`.

The first provides the benchmark-facing submission interface, while the second contains the reasoning and planning logic used to construct and evaluate candidate actions.

Role of the Reasoning-Based Controller

The current reasoning-based controller provides a practical decision mechanism for exercising the benchmark before a stronger trained policy is available. Rather than attempting exhaustive exploration of the encoded action space, it constructs a limited set of candidate semantic actions and compares selected candidates through environment feedback.

As discussed in Section 3.6, limiting the candidate set reduces the number of actions considered for backend trial evaluation, but it does not remove the cost of evaluating each selected candidate. The reasoning controller should therefore be interpreted as a practical mechanism for controlled candidate evaluation rather than as a general solution to sample-efficient reinforcement learning.

This controller should not be interpreted as a replacement for reinforcement learning. The environment remains RL-compatible, and the distinction lies only in how the next action is selected. A future trained policy can replace the current reasoning controller without changing the benchmark harness, semantic action representation, backend runner, or QoR evaluation path.

Candidate Semantic Actions

The reasoning layer operates on semantic actions rather than directly invoking backend scripts. Candidate transformations are therefore expressed using the same action abstraction introduced in Section 3.4.

This is important because it preserves the separation between decision making and physical execution. The controller can reason about an operation such as buffer insertion without needing to know how the corresponding Verilog transformation or OpenROAD invocation is implemented.

Candidate generation is also constrained by the benchmark task. The current validated task permits only buffer insertion, so the reasoning controller should not be interpreted as selecting freely among all transformation families contained in the repository. Candidate actions in the current experiment remain within the semantic action scope permitted by `TASK_0013`.

The broader framework can represent additional operations, but expanding the candidate set requires corresponding benchmark validation and backend support.

Rollback-Based Trial Evaluation

The key mechanism that enables planning-style evaluation is environment state restoration. The environment provides save-and-restore functionality that allows the controller to test a candidate action without permanently committing the resulting design state.

Suppose the environment is in state s_t and the reasoning controller generates a set of candidate actions

$$\mathcal{A}_{\text{cand}} = \left\{ a_t^{(1)}, a_t^{(2)}, \ldots, a_t^{(K)} \right\}. \tag{3.13}$$

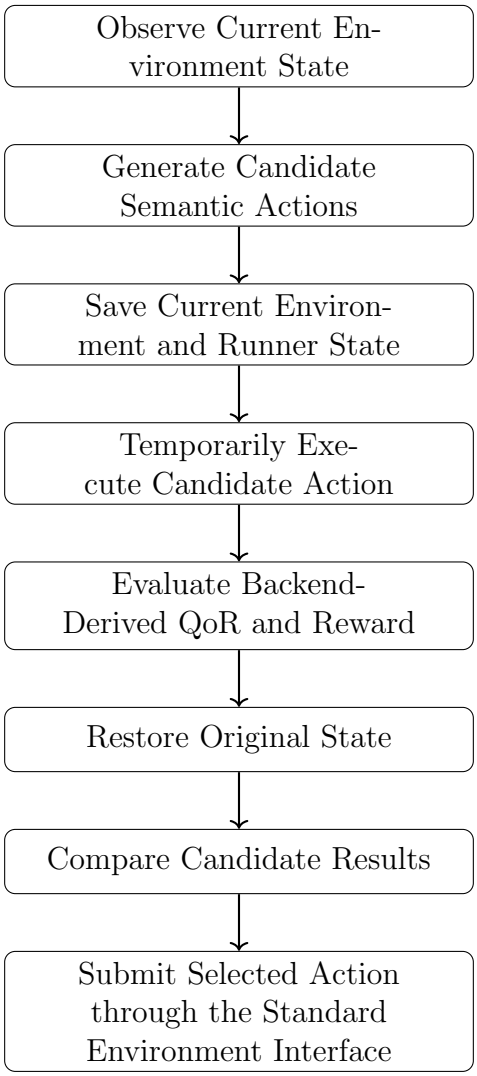

Figure 3.6: Reasoning-based candidate evaluation in the current RL-CHIP-Bench submission. Candidate semantic actions can be temporarily evaluated using the same environment and backend used for normal execution. State restoration allows multiple candidate actions to be compared before the selected action is committed.

Before candidate evaluation, the current environment and runner state are saved. For each candidate $a_t^{(i)}$, the controller temporarily executes the action through the normal environment interface and observes the resulting reward and QoR measurements:

$$\left(o_{t+1}^{(i)}, r_t^{(i)}\right) = \mathcal{E}\left(s_t, a_t^{(i)}\right). \tag{3.14}$$

After the trial result is recorded, the original state is restored before the next candidate is tested. This prevents one trial action from changing the starting point used to evaluate another candidate.

The candidate evaluations therefore share the same reference state, making their measured effects easier to compare.

Algorithm 3 Rollback-Based Candidate Evaluation

Require: Environment $\mathcal{E}$, current observation o_t
Ensure: Selected action a_t^*

$\mathcal{C} \leftarrow \textsc{GenerateCandidates}(o_t)$
$S_t \leftarrow \textsc{SaveState}(\mathcal{E})$
$\mathcal{R} \leftarrow \emptyset$
for all $a^{(i)} \in \mathcal{C}$ **do**
    $\textsc{RestoreState}(\mathcal{E}, S_t)$
    $(o^{(i)}, r^{(i)}, \text{terminated}, \text{truncated}, \text{info}^{(i)}) \leftarrow \textsc{Step}(\mathcal{E}, a^{(i)})$
    $\mathcal{R} \leftarrow \mathcal{R} \cup \left\{\left(a^{(i)}, r^{(i)}, \text{info}^{(i)}\right)\right\}$
end for
$\textsc{RestoreState}(\mathcal{E}, S_t)$
$a_t^* \leftarrow \textsc{SelectCandidate}(\mathcal{R})$
return a_t^*

Algorithm 3 summarizes the rollback-based candidate evaluation mechanism. Each candidate is evaluated from the same saved design state, and the environment is restored after

every trial. After the candidate results have been collected, the controller selects one action and returns it to the benchmark interaction loop.

The selected action is then executed normally by the benchmark environment. In other words, trial evaluation and committed execution use the same semantic action and backend interfaces; the difference is that trial states are restored while the final selected action is allowed to advance the episode.

Interaction with the Standard Environment

The reasoning controller does not bypass the RL-compatible environment. Candidate trials still use the same action decoding, task restriction, backend execution, QoR evaluation, and reward computation described in Sections 3.3–3.6.

This means that a candidate cannot gain access to an optimization operation that is forbidden by the benchmark simply because it is generated internally by the reasoning module. Task restrictions remain enforced by the environment during trial execution.

The same principle applies to backend evaluation. A candidate is not considered useful only because the reasoning layer predicts that it should improve the design. Its measured effect must still be obtained through the backend execution path.

This architectural separation keeps the controller interchangeable. The environment defines the optimization problem and execution semantics; the agent defines only the strategy used to select the next action.

Why Rollback Is Useful for EDA Optimization

Rollback is particularly useful in the current EDA setting because local transformations can have effects that are difficult to predict from the compact observation alone. A candidate buffer insertion may be syntactically valid while producing no timing improvement, or it may degrade the design after backend evaluation.

Without rollback, every candidate trial would permanently modify the working design. Subsequent candidates would then be evaluated from different starting points, making it difficult to distinguish the effect of one candidate from the accumulated effect of earlier trials.

State restoration instead supports controlled local comparison. Multiple candidate actions can be evaluated from the same design state before one action is selected for committed execution.

The disadvantage is computational cost. Every trial that reaches the real backend may require another OpenROAD evaluation. Candidate generation must therefore remain selective; exhaustive trial evaluation over a very large action space would be impractical. This trade-off is one reason why better target ranking and surrogate models remain important future directions.

Relationship to Reinforcement Learning

The current reasoning controller and a future trained RL policy differ primarily in their decision-making mechanism.

A trained reinforcement-learning policy would learn a mapping from observations to actions through accumulated interaction data and reward feedback. The current controller instead uses explicit candidate construction and trial evaluation at decision time.

However, both controllers interact with the same environment abstraction:

$$o_t \rightarrow a_t \rightarrow \text{backend} \rightarrow r_t, o_{t+1}. \tag{3.15}$$

This common interface is one of the intended properties of RL-CHIP-Bench. The benchmark infrastructure does not require a submission to use one particular learning algorithm. Instead, it defines a controlled optimization task under which different decision-making approaches can later be evaluated.

Current Limitations

The current reasoning submission should be interpreted primarily as a controller for validating the integrated framework rather than as a mature timing-optimization algorithm.

First, the validated benchmark task exposes only buffer insertion, so the current reasoning process does not demonstrate coordinated optimization across the full transformation library. Second, the compact QoR observation provides limited information for identifying the most

promising circuit targets. Global timing metrics can indicate whether the design improved, but they do not directly identify which local instance should be modified next.

Third, rollback-based trial evaluation increases backend cost because each candidate may require a real tool invocation. The approach therefore becomes less practical as the candidate set grows.

Finally, the current experiments do not demonstrate consistently strong timing improvement across a broad benchmark suite. The primary validated result is that the reasoning submission can operate through the same benchmark, environment, semantic action, and OpenROAD execution stack intended for future optimization agents.

Despite these limitations, the reasoning submission provides an important system-level validation of RL-CHIP-Bench. It demonstrates that the framework supports not only one-step agent interaction, but also temporary candidate evaluation, backend state restoration, and planning-style decision making without modifying the external benchmark protocol.

The next section presents the experimental configuration used to validate this complete execution path and distinguishes the infrastructure capabilities that have been demonstrated from the optimization capabilities that remain under development.

3.8 Experimental Setup and Validation

The experimental evaluation in this thesis focuses on validating the complete RL-CHIP-Bench execution path under a real OpenROAD-backed benchmark task. The goal of the

current experiment is to determine whether the benchmark harness, environment, semantic action layer, backend runner, reasoning-based submission, QoR evaluation, and artifact pipeline can operate together as one controlled system.

The current experiment is not intended to establish state-of-the-art timing optimization performance. The validated task exposes a narrow action subset, and the current reasoning-based controller is still under development. The evaluation therefore distinguishes between *system validation*, which has been demonstrated, and *optimization quality*, which remains limited in the current implementation.

The active experimental flow is

`TASK_0013` → `arc_reasoning` → RL-CHIP-Bench environment → OpenROAD → benchmark artifacts.

Experimental Configuration

The current validated experiment uses an AES gate-level design with the Nangate45 technology configuration and OpenROAD as the QoR evaluation backend.

The active design and technology resources are summarized in Table 3.3.

Table 3.3: Experimental configuration of the current validated RL-CHIP-Bench run.

Component	Configuration
Benchmark task	`TASK_0013`
Design	AES gate-level design
Technology / PDK	Nangate45
Backend	OpenROAD
Submission	Reasoning-based agent (`arc_reasoning`)
Allowed transformation	Buffer insertion
Design configuration	`rl/configs/design_aes_nangate45.yaml`
PDK configuration	`rl/pdks/nangate45.yaml`
Gate-level netlist	`../aes_project/aes/rtl/aes_nangate45.v`
Timing constraints	`../aes_project/aes/sdc/aes_nangate45.sdc`
QoR evaluation	OpenROAD timing analysis

Validation Objectives

The current experiment evaluates the integration of the framework at several levels.

1. **Task-level validation.** The benchmark harness must successfully load the task specification and resolve the required design, technology, and backend resources.

2. **Submission-level validation.** The reasoning-based submission must be accepted by the benchmark harness and interact with the environment through the standard agent interface.

3. **Environment-level validation.** The environment must initialize correctly, provide observations, decode submitted actions, enforce task-specific restrictions, and maintain the episode state.

4. **Backend-level validation.** A valid semantic action must be translated into a concrete design modification that can be evaluated successfully through the configured OpenROAD flow.

5. **QoR-feedback validation.** The backend result must be parsed into structured QoR measurements that can be returned to the environment and used for reward computation.

6. **Artifact-level validation.** The benchmark run must produce structured output artifacts that record the final experimental state and can later support quantitative comparison.

These objectives are intentionally system-oriented. Successful completion of all six levels demonstrates that the framework can execute a complete EDA-backed optimization episode, but it does not by itself demonstrate that the current policy consistently improves timing.

End-to-End Validation Result

The current RL-CHIP-Bench implementation successfully completes the full benchmark execution path for `TASK_0013`. The task can be loaded by the benchmark harness, the `arc_reasoning` submission can be instantiated, and the environment can interact with the real backend through the configured semantic action interface.

During execution, the environment accepts benchmark-compatible actions and forwards valid operations to the production runner. The backend applies the corresponding netlist

transformation, invokes OpenROAD, and returns parsed QoR measurements to the environment. The benchmark then records the resulting execution information and final metrics in structured artifacts.

This result demonstrates that the following components are operational together:

- benchmark task loading and validation;
- submission integration;
- RL-compatible environment interaction;
- semantic action decoding and task restriction;
- real backend execution;
- OpenROAD QoR evaluation;
- reward and environment-state update;
- structured artifact generation.

The successful end-to-end execution is the primary experimental validation of the current framework.

Table 3.4: Validation status of the current RL-CHIP-Bench execution path.

Validation item	**Current status**
Benchmark task loading	Validated
Task resource validation	Validated for `TASK_0013`
Environment construction	Validated
Semantic action decoding	Validated
Task-specific action restriction	Validated
Production runner execution	Validated
Real OpenROAD invocation	Validated
QoR parsing	Validated
Reasoning-agent integration	Validated
Structured artifact generation	Validated
Full benchmark-suite coverage	Not yet complete
Strong timing improvement	Not yet demonstrated

Experimental Artifacts

The validated run produces structured benchmark outputs rather than relying only on console logs. The two primary artifacts currently used for analysis are

`benchmark/runs/TASK_0013/arc_reasoning_seed0/run_summary.json`

and

`benchmark/runs/TASK_0013/arc_reasoning_seed0/artifacts/final_metrics.json`.

The first records run-level execution information, while the second stores the final QoR-related measurements produced by the benchmark flow. These files provide persistent evidence of the completed experiment and can be processed automatically in later comparisons.

The use of structured artifacts is especially important for future benchmark expansion. When multiple agents or tasks are evaluated, their results can be compared using the same

artifact schema instead of manually extracting values from terminal output. This supports the reproducibility goals discussed in Chapter 2 [6, 2].

Measured QoR Results

The validated backend run provides quantitative QoR measurements that can be used to evaluate the effect of the executed transformations. Table 3.5 summarizes the measurements extracted from the available OpenROAD evaluation artifacts.

The initial evaluated design has a worst negative slack (WNS) of 0.048775 ns and a total negative slack (TNS) of 0. The positive WNS indicates that the reported setup timing constraint is satisfied at this point. The corresponding power measurement is approximately 1.560197 mW.

After the evaluated transformation sequence, the reported WNS decreases to 0.025528 ns while TNS remains 0. The design therefore continues to satisfy the reported timing constraint, but the available positive timing margin is reduced. The final reported power increases to approximately 1.582045 mW.

Table 3.5: Measured QoR values from the available OpenROAD-backed evaluation sequence.

Metric	**Initial**	**Final**	**Change**
WNS (ns)	0.048775	0.025528	−0.023248
TNS (ns)	0.000000	0.000000	0.000000
Power (mW)	1.560197	1.582045	+0.021848

The WNS change corresponds to a reduction of approximately 0.02325 ns in positive timing margin. Relative to the initial WNS value, this is a decrease of approximately 47.7%.

Because both the initial and final WNS values remain positive and TNS remains zero, the evaluated design still satisfies the reported timing constraint.

Power increases by approximately 0.02185 mW, or about 1.40%, over the same evaluation sequence. The available measurements therefore do not indicate a QoR improvement for this particular transformation sequence. Instead, they demonstrate that RL-CHIP-Bench is able to detect and report both timing degradation and power overhead produced by real design modifications.

This result is useful for validating the optimization loop because the backend is not biased toward reporting a successful action as an improvement. A syntactically valid transformation can still receive an unfavorable QoR evaluation when its measured effect is negative.

Table 3.6: QoR progression across the available OpenROAD evaluation records.

Evaluation state	**WNS (ns)**	**TNS (ns)**	**Power (mW)**
Initial	0.048775	0.000000	1.560197
Intermediate	0.025528	0.000000	1.574687
Later evaluation	0.025528	0.000000	1.582045
Final recorded state	0.025528	0.000000	1.582045

Table 3.6 shows that the largest timing change occurs between the initial state and the first modified state. Subsequent evaluations preserve the same reported WNS and TNS while power increases further. This suggests that the later evaluated modifications did not provide additional timing benefit under the current backend configuration.

The repeated final QoR values also provide a useful consistency check: equivalent or

unchanged evaluated states produce the same timing result, while the power records reflect the corresponding modified design state.

Evidence of Real Netlist Modification

The timing reports provide additional evidence that the evaluated QoR values correspond to a modified gate-level design rather than to a mock environment transition.

In the modified timing path, OpenROAD reports an inserted buffer instance named `BUF_INS_modified_1` using a `BUF_X8` cell. The instance appears directly on the reported maximum-delay path before the destination flip-flop.

The corresponding path remains timing-clean, with a reported slack of approximately 0.026 ns. Compared with the initial reported maximum-path slack of approximately 0.049 ns, however, the available timing margin is smaller after the modification.

This report-level evidence confirms that the benchmark action is translated into an actual gate-level transformation that is subsequently visible to OpenROAD timing analysis.

Interpretation of Current Optimization Quality

The current experiment demonstrates successful system integration, but it does not yet demonstrate strong timing optimization by the reasoning-based controller.

The benchmark task is deliberately narrow and currently validates buffer insertion as the real optimization action. In addition, the controller operates from a compact global QoR

observation and does not yet use detailed critical-path or graph-based target information. As a result, the system can execute and evaluate real candidate transformations, but the quality of target selection remains limited.

This distinction is important when interpreting the experimental result. A valid action and a successful backend run demonstrate that the execution infrastructure is functioning, but they do not imply that the selected transformation improves timing.

The current result therefore supports the system-level claim of this thesis: RL-CHIP-Bench provides an operational, benchmark-integrated path from agent decision making to real OpenROAD evaluation. It does not yet support a claim of consistently improved WNS or TNS across a broad set of benchmark designs.

Scope of the Validation

The current validation is limited to one real benchmark task in the active workspace. The full benchmark suite has not yet been retargeted and validated under the same environment.

Some additional tasks may require design-resource retargeting, path correction, or task-specific backend configuration before they can be executed reliably. Similarly, the broader transformation library has not been validated uniformly under the current real benchmark task.

For this reason, the current experiment should be interpreted as an end-to-end system validation rather than a broad benchmark study. Demonstrating scalability across multiple

designs, technologies, random seeds, and agent types remains future work.

Overall, the experiment verifies that the principal RL-CHIP-Bench components can operate together under a real EDA-backed task. The framework successfully connects task specification, benchmark validation, agent interaction, semantic action execution, OpenROAD-based QoR evaluation, reward feedback, and structured result logging.

The next section discusses the implementation challenges and current limitations that became apparent while integrating these components into one executable research system.

3.9 Implementation Challenges and Limitations

Integrating reinforcement-learning interfaces with a real EDA backend introduced several engineering challenges that were less visible in the initial algorithm-level formulation. These challenges affected not only software debugging, but also the architecture of the final RL-CHIP-Bench implementation.

The most important issues involved toolchain reproducibility, backend resource management, action validity, state consistency, execution cost, and the coexistence of earlier prototypes with the current benchmark-integrated implementation. This section summarizes these issues and the design decisions that resulted from them.

Toolchain and Dependency Management

An early stage of the project explored graph-based circuit representations using DGL and GraphSAGE. During this work, compatibility problems between DGL, PyTorch, and GraphBolt prevented the graph-processing pipeline from running reliably under some software configurations.

The practical solution during development was to use a compatible CPU-based DGL configuration so that graph extraction and embedding experiments could proceed. This experience highlighted a broader reproducibility issue: an ML-for-EDA system depends simultaneously on machine-learning libraries and EDA tools, and incompatibility in either software stack can prevent an otherwise valid experiment from running.

For the current RL-CHIP-Bench implementation, graph processing is therefore not placed on the critical execution path. The validated benchmark uses a compact QoR observation, while graph-based state representations remain an extension rather than a mandatory runtime dependency.

This decision reduces the dependency burden of the current benchmark and allows the environment, action representation, and OpenROAD backend to be validated independently of a specific graph-learning software stack.

Backend Resource and Path Management

Real OpenROAD execution requires several design and technology resources to be resolved consistently. These include the gate-level netlist, timing constraints, LEF data, Liberty libraries, top-module information, and backend scripts.

During development, hard-coded or prototype-specific paths created a risk that one script could operate on resources different from those intended by the benchmark task. Such errors are particularly difficult to identify because the backend may fail only after several stages of tool execution.

The current benchmark-oriented design addresses this problem by moving resource selection into explicit task and design configuration files. The benchmark harness resolves these resources before constructing the runtime environment, and the backend receives the resolved configuration rather than independently selecting local files.

This change is one reason the task manifest and validation layers described in Section 3.5 are important. They reduce the number of implicit assumptions that must remain synchronized across Python, Tcl, and Verilog-processing components.

QoR Report Parsing and Interface Stability

Another integration challenge is the conversion of EDA-tool output into a stable machine-learning interface. OpenROAD produces timing and design information through tool-specific reports and execution outputs, whereas the environment expects structured numerical values

such as WNS and TNS.

Directly parsing tool output inside the environment would tightly couple the RL interface to one particular report format. It would also make debugging difficult because changes to the backend output could appear as failures in unrelated environment code.

The current implementation therefore isolates report interpretation in the QoR evaluator. The runner receives structured measurements from this layer rather than parsing raw OpenROAD output itself.

This separation does not eliminate all backend-format dependencies, but it confines them to one module. If the OpenROAD reporting format or evaluation script changes, the parsing layer can be updated without redesigning the agent–environment interface.

Action Validity and Target Resolution

A fixed-dimensional action interface simplifies agent integration, but it does not guarantee that every encoded action corresponds to a useful physical transformation.

Target bins may be empty, multiple circuit objects may collide in the same bin, and a syntactically valid operation may still be unsuitable for a particular design object. Task-level restrictions introduce another layer of validity because an action represented by the global environment may not be permitted by the active benchmark task.

The current implementation therefore treats action decoding, task permission, target resolution, and backend execution as separate stages. An action can be rejected before

backend execution when it is incompatible with the task, while backend failures remain distinguishable from task-level rejection.

This distinction is useful both for debugging and for future agent training. If all unsuccessful actions were represented only by one generic failure signal, the agent would receive little information about whether the problem originated from the task definition, target mapping, or physical execution.

Maintaining Design-State Consistency

State consistency becomes especially important when an agent is allowed to evaluate temporary candidate transformations. A backend state includes more than the scalar QoR values visible to the agent; it also includes the actual working design on which subsequent transformations will operate.

Restoring only the observation vector would therefore be insufficient. After a temporary candidate is evaluated, both the environment state and the backend design state must return to the same pre-trial condition before another candidate is tested.

The save-and-restore mechanism used by the reasoning controller was introduced to enforce this requirement. It allows multiple candidates to be evaluated from a common reference state and prevents trial actions from accumulating unintentionally.

This mechanism also revealed an important architectural requirement: planning support cannot be implemented only inside the agent. The runner and environment must expose

compatible state-management behavior for rollback to be reliable.

Cost of Real EDA Interaction

Real backend evaluation substantially increases the cost of an environment transition. A single step may require netlist editing, file generation, OpenROAD startup and execution, timing analysis, report parsing, and artifact handling.

This has a direct effect on reinforcement-learning design. Exploration strategies that are reasonable in inexpensive simulation environments can become impractical when thousands of actions each require a real EDA invocation.

The current reasoning-based controller partially addresses this issue by limiting candidate generation rather than attempting exhaustive exploration. However, rollback-based planning can itself increase execution cost because several trial actions may be evaluated before one action is committed.

For future RL training, improving sample efficiency will therefore be important. Candidate filtering, graph- or timing-aware target ranking, cached evaluations, offline experience, and surrogate QoR prediction are possible approaches to reducing unnecessary backend calls. Real OpenROAD evaluation should nevertheless remain the final source of verification for reported design quality.

Transition from Prototype Scripts to Benchmark Infrastructure

A separate challenge during the project was maintaining a clear distinction between early experimental prototypes and the current research system.

Earlier stages included graph-embedding experiments, lightweight QoR prediction, and a standalone DQN-style OpenROAD prototype. These implementations were useful for testing individual ideas, but they did not provide the task validation, agent-independent interfaces, structured artifacts, and reproducibility controls required by the final benchmark-oriented design.

As RL-CHIP-Bench developed, retaining both older scripts and newer benchmark modules in the workspace created a risk of referring to an obsolete execution path when documenting or running an experiment. For example, an older script might invoke OpenROAD correctly without passing through the current benchmark harness or semantic action interface.

The thesis therefore treats the benchmark-integrated path as the authoritative implementation:

$$\text{Task Specification} \rightarrow \text{Benchmark Harness} \rightarrow \text{Environment} \rightarrow \text{Semantic Action} \rightarrow$$
$$\texttt{ResynthesisRunner} \rightarrow \text{OpenROAD}.$$

Earlier prototypes are retained only as supporting development work and are not used as the main evidence for the experimental claims in this thesis.

Current Limitations

The current implementation has several limitations that define the scope of the conclusions in this thesis.

First, the real benchmark validation currently focuses on one task, one AES design, and the Nangate45 technology configuration. The framework architecture is intended to support additional tasks and designs, but those configurations have not yet been validated to the same extent.

Second, the currently validated real transformation is buffer insertion. Although the repository contains semantic representations and implementation support for additional transformation families, they should not be treated as experimentally validated under the current benchmark.

Third, the compact QoR observation provides only global design information. It does not directly expose the critical paths, local timing context, or graph structure that would allow an agent to identify high-quality optimization targets more precisely.

Fourth, the current reasoning-based controller has not demonstrated consistently strong timing improvement. Its main value in the present experiments is the validation of planning-style interaction, candidate trial execution, rollback, and compatibility with the real backend.

Fifth, real OpenROAD interaction limits the number of actions that can be evaluated efficiently. Scaling the framework to substantial RL training will require better action selection and more sample-efficient use of backend evaluations.

These limitations do not invalidate the system-level results presented in this chapter, but they restrict the conclusions that can be drawn about optimization performance. The current thesis demonstrates an operational research framework and validated execution path; a broader performance study remains future work.

Table 3.7: Major implementation challenges and corresponding design responses.

Challenge	**Design response**
ML library compatibility	Keep graph processing outside the required real benchmark execution path.
Design and technology paths	Resolve resources through explicit task and design configuration.
EDA report formats	Isolate OpenROAD output parsing in the QoR evaluator.
Variable and invalid targets	Separate action encoding, target resolution, task validation, and backend execution.
Planning-state consistency	Save and restore both environment and runner state during candidate trials.
Expensive backend evaluation	Limit candidate trials and retain sample efficiency as a primary future requirement.
Legacy and current code paths	Treat the benchmark-integrated RL-CHIP-Bench path as the authoritative experimental flow.

3.10 Chapter Summary

This chapter presented the design, implementation, and validation of RL-CHIP-Bench, a benchmark-integrated framework for sequential chip-design optimization with a real OpenROAD backend.

The framework separates benchmark control, agent interaction, semantic action representation, backend execution, QoR evaluation, and result logging into distinct software layers.

This separation allows different decision-making methods to reuse the same task definition and backend infrastructure without directly depending on design-specific scripts.

The environment exposes a reinforcement-learning-compatible sequential interface while hiding the details of netlist modification and OpenROAD execution. A fixed-dimensional action representation is converted into semantic design operations and then mapped to design-dependent targets. The benchmark harness controls task configuration, action restrictions, resource validation, runtime construction, scoring, and artifact generation.

The backend execution path connects semantic actions to real design modifications and OpenROAD-based QoR evaluation. In the current validated task, buffer insertion is used as the real transformation, and timing-related measurements such as WNS and TNS are returned to the environment for reward computation and later analysis.

The current benchmark also demonstrates compatibility with a reasoning-based submission. Through environment state save-and-restore, the controller can evaluate temporary candidate actions from a common design state before selecting an action for committed execution. This shows that the same infrastructure can support planning-style controllers in addition to future reinforcement-learning agents.

The experimental validation confirms that the benchmark harness, environment, semantic action layer, production runner, OpenROAD backend, QoR parser, reasoning-based submission, and artifact pipeline can operate together under one controlled task configuration. However, the current results should be interpreted as system-level validation rather than as evidence of mature timing optimization. The validated experiment is limited to one bench-

mark task, one AES design, the Nangate45 technology configuration, and buffer insertion as the real optimization action.

Several limitations remain. The compact QoR observation provides limited local timing information, the target-bin mapping introduces approximate target resolution, real back-end evaluation is computationally expensive, and the current reasoning controller has not yet demonstrated consistently strong timing improvement. Extending the framework to additional designs, transformations, graph-based observations, and trained reinforcement-learning policies therefore remains an important direction for future work.

Overall, the main contribution of this chapter is the implementation and validation of an operational research infrastructure that connects agent decision making with real EDA execution under a reproducible benchmark protocol. The next chapter summarizes the overall thesis contributions and discusses future extensions of the framework.

Chapter 4

Conclusion

4.1 Summary of the Work

This thesis investigated the infrastructure required to connect sequential decision-making methods with real chip-design optimization flows. The work was motivated by the observation that applying reinforcement learning or other learning-based methods to EDA requires more than defining a state, action, and reward function. The optimization method must ultimately interact with real circuit representations, valid transformations, technology resources, timing constraints, backend tools, and reproducible evaluation procedures.

To address these requirements, this thesis developed and validated the RL-CHIP-Bench system described in Chapter 3. The framework separates benchmark control, environment interaction, semantic action representation, backend execution, QoR evaluation, reward computation, and agent decision making into distinct components.

Chapter 2 introduced the technical foundations required by the system, including reinforcement learning for sequential optimization, timing-oriented digital design optimization, graph representations of circuits, and reproducible ML-for-EDA infrastructure. These topics provide the background for the implementation without organizing the thesis around individual graduate courses.

Chapter 3 described the architecture and implementation of RL-CHIP-Bench. The benchmark harness defines controlled optimization tasks, while the RL-compatible environment exposes observations, actions, rewards, and episode behavior through a standardized sequential interface. A semantic action layer separates the fixed-dimensional agent representation from design-dependent physical targets. The backend runner translates these actions into actual gate-level transformations and invokes OpenROAD to obtain measured QoR feedback.

The current validated implementation also includes a reasoning-based controller that uses temporary candidate execution and state restoration. This demonstrates that the same environment can support planning-style decision making without embedding agent-specific logic into the benchmark or EDA backend.

4.2 Main Contributions

The main contributions of this thesis are summarized as follows.

1. **A benchmark-integrated architecture for chip-design optimization.** RL-CHIP-Bench separates task specification, benchmark control, agent interaction, backend ex-

ecution, scoring, and artifact logging so that optimization methods can be evaluated under a common experimental protocol.

2. **An RL-compatible environment connected to a real EDA backend.** The environment exposes a sequential observation–action–reward interface while delegating physical design modification and QoR evaluation to the backend. This separation allows different controllers to reuse the same execution infrastructure.

3. **A semantic action abstraction for design-dependent transformations.** A fixed-dimensional agent action is decoded into a semantic operation and then resolved to a concrete circuit target. This avoids directly coupling the agent action space to instance names or design-specific backend scripts.

4. **Real OpenROAD-based QoR evaluation.** Valid semantic actions are translated into gate-level netlist modifications and evaluated through OpenROAD. The resulting timing and other available QoR measurements are returned to the environment and preserved in structured benchmark artifacts.

5. **Task-level action control and reproducible configuration.** Benchmark manifests define design resources, technology configuration, action restrictions, episode settings, and evaluation parameters independently of the agent.

6. **Support for reasoning-based candidate evaluation with rollback.** Environment and runner state can be saved and restored, allowing candidate transformations to be

evaluated from a common design state before one action is committed.

7. **Structured experimental logging and result generation.** The benchmark produces persistent machine-readable artifacts rather than relying on terminal output, providing a foundation for future multi-agent and multi-task comparisons.

Together, these contributions establish an executable connection between high-level agent decision making and low-level EDA evaluation. The contribution is primarily architectural and system-oriented rather than the introduction of a new reinforcement-learning algorithm.

4.3 Experimental Findings

The experimental work focused on validating the complete execution path of the framework using a real OpenROAD-backed configuration. The validated flow demonstrates successful task construction, agent integration, semantic action execution, netlist modification, OpenROAD timing analysis, QoR parsing, reward feedback, and structured artifact generation.

The available OpenROAD measurements also provide quantitative evidence that the backend responds to actual design changes. In the evaluated sequence, the initial WNS is approximately 0.0488 ns and TNS is 0, indicating that the reported setup timing constraint is satisfied. After the evaluated transformations, WNS decreases to approximately 0.0255 ns while TNS remains 0. The design therefore remains timing-feasible, but its positive timing margin is reduced.

The measured power increases from approximately 1.5602 mW to 1.5820 mW, corresponding to an increase of approximately 1.40%. The available sequence therefore does not demonstrate an overall QoR improvement. Instead, it provides an important validation of the framework: a successfully executed action is not automatically classified as a successful optimization. The measured backend result determines whether the transformation improves or degrades the design.

The timing reports also show the inserted buffer directly on the analyzed timing path, providing evidence that the agent-facing semantic action is translated into an actual gate-level modification before OpenROAD evaluation.

These results support the system-level contribution of the thesis, but they do not support a claim of mature timing optimization performance. A broader experimental study would require additional benchmark designs, transformations, agents, random seeds, and technology configurations.

4.4 Limitations

The current framework has several limitations.

First, real backend validation currently covers a limited experimental scope. The main validated configuration uses one benchmark task, an AES gate-level design, the Nangate45 technology setup, and buffer insertion as the real transformation. Additional designs and transformation families have not yet been validated to the same extent.

Second, the current observation is based mainly on compact global QoR measurements. Although WNS, TNS, power, and related metrics describe the overall state of the design, they provide limited information about which local instance or path should be modified. This restricts the quality of target selection.

Third, the fixed target-bin representation provides a stable action interface but introduces empty-bin and collision problems. More informed target generation will be needed for stronger optimization performance.

Fourth, real EDA evaluation is computationally expensive. Each candidate action may require netlist processing, OpenROAD execution, timing analysis, and report parsing. This makes large-scale online exploration more expensive than in conventional simulation environments.

Fifth, the current reasoning-based controller has not demonstrated consistently strong QoR improvement. Its primary role in this thesis is to validate planning-style interaction, candidate evaluation, rollback, and compatibility with the real backend.

Finally, the current evaluation does not yet include a complete multi-design, multi-seed comparison against mature timing-optimization baselines. The results should therefore be interpreted as validation of an operational research framework rather than as evidence of superiority over existing EDA optimization methods.

4.5 Future Work

Several extensions are natural next steps for RL-CHIP-Bench.

Trained Reinforcement-Learning Agents

The first direction is to replace or complement the current reasoning controller with trained reinforcement-learning policies. The environment interface is already structured around sequential observations, actions, rewards, and episode boundaries, allowing methods such as value-based or actor–critic agents to interact with the same backend without modifying the benchmark architecture [15, 13, 14].

A practical challenge will be sample efficiency. Because real OpenROAD evaluations are expensive, future RL experiments should minimize unnecessary backend calls and make effective use of previously collected transitions.

Graph-Based Observations and Target Selection

A second direction is to incorporate local circuit structure into the observation and action selection process. Gate-level netlists naturally admit graph representations, and graph neural networks can aggregate structural information across connected circuit objects [5].

Graph features could be used to identify timing-critical regions, rank candidate targets, or provide embeddings to an RL policy. This would address a major limitation of the current global QoR observation, which provides information about whether timing improved but

little information about where the next useful transformation should be applied.

Expanded Transformation Library

The current validated task focuses on buffer insertion. Future benchmark tasks should progressively validate additional transformations such as buffer deletion, gate resizing, threshold-voltage swapping, gate cloning, and local logic restructuring.

A larger action set would create a more realistic sequential optimization problem because different transformations produce different timing, power, and area trade-offs. However, each operation should first be validated independently before being combined into a larger agent action space.

Surrogate and Partial QoR Models

Another direction is to introduce a lightweight predictive model before expensive backend evaluation. A surrogate could rank candidate actions or reject low-quality candidates before OpenROAD is invoked.

The earlier graph and partial-model experiments in this project suggest one possible path for this extension. The surrogate should not replace real EDA evaluation as the final source of QoR, but it could reduce the number of backend calls required during exploration or planning.

Broader Benchmark Evaluation

The current system should also be extended to additional circuits, task configurations, technology libraries, agents, and random seeds. A broader benchmark suite would allow the framework to move from end-to-end system validation toward quantitative comparison of optimization methods.

The benchmark-oriented design of RL-CHIP-Bench is intended to support this transition: task manifests, standardized environment interfaces, action restrictions, scoring, and structured artifacts can remain consistent while the evaluated designs and agents change.

4.6 Final Remarks

This thesis demonstrates that learning-guided chip optimization is not only an algorithmic problem, but also an infrastructure problem. Before an optimization agent can be evaluated meaningfully, the system must establish reliable connections among task definitions, circuit representations, valid transformations, backend execution, QoR measurements, reward feedback, and reproducible experimental records.

RL-CHIP-Bench provides such a connection in the current validated implementation. The framework does not yet solve timing optimization across a broad benchmark suite, and the current controller does not yet demonstrate consistently strong QoR improvement. However, it establishes an operational path from agent-level decisions to real gate-level modification and OpenROAD evaluation.

This infrastructure provides a foundation on which stronger learning, reasoning, and hybrid optimization methods can be developed and evaluated under a common EDA-backed benchmark protocol.

Bibliography

[1] Lilas Alrahis et al. "GNN-RE: Graph Neural Networks for Reverse Engineering of Gate-Level Netlists". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021). DOI: 10.1109/TCAD.2021.3110807.

[2] Vidya A. Chhabria et al. "Invited: Toward an ML EDA Commons: Establishing Standards, Accessibility, and Reproducibility in ML-Driven EDA Research". In: *Proceedings of the 2025 International Symposium on Physical Design (ISPD)*. 2025, pp. 1–9. DOI: 10.1145/3698364.3709131.

[3] Vidya A. Chhabria et al. "OpenROAD and CircuitOps: Infrastructure for ML EDA Research and Education". In: *Proceedings of the 2024 IEEE 42nd VLSI Test Symposium (VTS)*. 2024.

[4] Yufan Du et al. "Differentiable Physical Optimization". In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2025.

[5] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.

[6] Shengyi Huang et al. *Open RL Benchmark: Comprehensive Tracked Experiments for Reinforcement Learning*. 2024. arXiv: 2402.03046 [cs.LG].

[7] Stephen James et al. "RLBench: The Robot Learning Benchmark and Learning Environment". In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 3019–3026. DOI: 10.1109/LRA.2020.2974707.

[8] Wenjing Jiang, Vidya A. Chhabria, and Sachin S. Sapatnekar. "IR-Aware ECO Timing Optimization Using Reinforcement Learning". In: *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*. 2024. DOI: 10.1145/3670474.3685945.

[9] Yi-Chen Lu et al. "RL-Sizer: VLSI Gate Sizing for Timing Optimization Using Deep Reinforcement Learning". In: *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 733–738. DOI: 10.1109/DAC18074.2021.9586138.

[10] Jianchao Lu and Baris Taskin. "From RTL to GDSII: An ASIC Design Course Development Using Synopsys® University Program". In: *2011 IEEE International Conference on Microelectronic Systems Education*. IEEE. 2011, pp. 72–75.

[11] Azalia Mirhoseini et al. "A Graph Placement Methodology for Fast Chip Design". In: *Nature* 594 (2021), pp. 207–212. DOI: 10.1038/s41586-021-03544-w.

[12] Azalia Mirhoseini et al. *Chip Placement with Deep Reinforcement Learning*. 2020. arXiv: 2004.10746 [cs.LG].

[13] Volodymyr Mnih et al. "Human-level Control through Deep Reinforcement Learning". In: *Nature* 518.7540 (2015), pp. 529–533. DOI: 10.1038/nature14236.

[14] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. Cambridge, MA: MIT Press, 2018.

[16] Vaibbhav Taraate. "ASIC Design and Synthesis". In: *Springer Nature* (2021).

[17] Christopher J. C. H. Watkins and Peter Dayan. "Q-Learning". In: *Machine Learning* 8.3–4 (1992), pp. 279–292. DOI: 10.1007/BF00992698.

[18] Hang Yang et al. *Pieceformer: Similarity-Driven Knowledge Transfer via Scalable Graph Transformer in VLSI*. 2025. arXiv: 2506.15907 [cs.LG].