Technische Universität München
TUM School of Computation, Information and Technology

On Using Machine Learning for Early Timing Estimation in Digital Design

Daniela Epple

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung einer

Doktorin der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:	Prof. Dr.-Ing Ralf Brederlow
Prüfende der Dissertation:	1. Hon.-Prof. Dr.-Ing Wolfgang Ecker
	2. Prof. Dr.-Ing Ulf Schlichtmann

Die Dissertation wurde am 13.09.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 26.10.2025 angenommen.

Abstract

The high demand for Application-Specific Integrated Circuits (ASICs) and the design productivity gap resulting from Moore's law motivate the Electronic Design Automation (EDA) field to embrace new technologies and accelerate the design flow, aiming at reducing design iterations or spins and, consequently, the time-to-market. One crucial aspect of an ASIC design is achieving timing closure. Unfortunately, feedback about the timing behavior of a Register-Transfer Level (RTL) design is delayed, as it involves different teams and tools for synthesis and timing analysis. Thus, timing violations are commonly encountered late in the flow and cause spins, potentially revisiting the initial RTL stage. Early timing information at RTL could enable earlier design choices and prevent violations later in the design flow. Classical methods for estimating timing from RTL designs rely on analytical or physical models and do not exploit the vast data produced by ETA tools. In contrast, Machine Learning (ML) models learn from those data and abstract the complexity of the design flow to approximate important design metrics. However, most ML-based estimations take a gate-level netlist as input and do not target optimization at the RTL stage.

This thesis proposes ML models as fast methods for estimating timing-related metrics at the RTL design stage. Training data are built using features of designs generated by a model-driven hardware generation framework. Ground-truth labels for delays, slews, arrival times, and slacks are extracted using open-source tools for logic synthesis and Static Timing Analysis (STA). Training datasets are built to represent the RTL designs using tabular and graph datasets. State-of-the-art ML architectures, such as graph neural networks, are also implemented. ML models are trained for three main tasks: estimating the delay of input-to-output connections for all components in an RTL Intermediate Representation (IR), estimating the worst-case path arrival time of the RTL IR, and finally, building a consolidated block-based timing estimation engine that annotates all pins with timing-related metrics. The ML-based flow provides sufficiently precise timing metrics for all pins in an RTL IR without the need for RTL generation, synthesis, or STA runs. Additionally, the main benefit of the proposed methods is that running ML inference is faster than running the EDA tools.

The timing estimations from this thesis pinpoint the RTL stage's critical regions, guiding engineers to explore different microarchitectures and coding styles without requiring even the RTL generation. Moreover, timing estimations are beneficial within RTL generation frameworks, where automated transformations generate variants of the same RTL design. Based on this estimations the best variant can be selected. A manual or automated microarchitecture search could lead to an optimal RTL IR, achieving timing closure faster and reducing the number of spins during the design flow. Thus, early ML-based timing estimations can shorten time-to-market. All methods in this thesis have been developed and evaluated within an industrial environment at Infineon Technologies AG.

Zusammenfassung

Die hohe Nachfrage nach anwendungsspezifischen integrierten Schaltungen (ASICs) und die aus dem Mooreschen Gesetz resultierende Produktivitätslücke im Design motivieren den Bereich der elektronischen Entwurfsautomatisierung (EDA), neue Technologien einzusetzen und den Designfluss zu beschleunigen, mit dem Ziel, Designiterationen oder Spins und damit die Zeit bis zur Markteinführung zu reduzieren. Ein entscheidender Aspekt eines ASIC-Entwurfs ist das Erreichen des Timing Closure. Dabei verzögert sich die Rückmeldung über das Timing-Verhalten eines Register-Transfer-Level (RTL)-Entwurfs, da verschiedene Teams und Tools für die Synthese und die Timing-Analyse involviert sind. Daher werden Timing-Verletzungen häufig erst spät im Designfluss festgestellt und führen zu der Notwendigkeit von wiederholten Designiterationen, bei denen die erste RTL-Phase möglicherweise erneut durchlaufen werden muss. Frühzeitige Timing-Informationen für RTL Entwürfe können frühere Designentscheidungen ermöglichen und spätere Verletzungen im Designfluss verhindern. Klassische Methoden zur Schätzung des Timings von RTL-Entwurfen beruhen auf analytischen oder physikalischen Modellen und nutzen die umfangreichen Daten, die von EDA-Tools erzeugt werden, nicht vollständig aus. Im Gegensatz dazu lernen Modelle des maschinellen Lernens (ML) aus diesen Daten und abstrahieren die Komplexität des Designflusses, um wichtige Designmetriken zu approximieren. Die gegenwärtig eingesetzten ML-basierten Schätzungen verwenden jedoch eine Gate-Level-Netzliste als Eingabe und zielen nicht auf eine Optimierung in der RTL-Phase ab.

Diese Dissertation schlägt ML-Modelle als schnelle Methoden zur Schätzung von timing-bezogenen Metriken in der RTL-Entwurfsphase vor. Trainingsdaten werden anhand von Merkmalen von bekannten Entwürfen aufgebaut, die von einem modellgesteuerten Hardware-Generierungs-Framework erzeugt wurden. Ground-Truth-Labels für Verzögerungen, Slews, Ankunftszeiten und Slacks werden mithilfe von Open-Source-Tools für logische Synthese und statische Timing-Analyse (STA) extrahiert. Die so erzeugten Trainingsdatensätze sind geeignet, um die RTL-Entwürfe anhand von tabellarischen und graphischen Datensätzen darzustellen, die wiederum geeignet für die Verwendung mit modernen ML-Architekturen, wie z.B. Graph-Neuronale Netzwerke, sind. Diese Arbeit präsentiert ML-Modelle für drei Hauptaufgaben: Schätzung der Verzögerung von Eingangs-zu-Ausgangs-Verbindungen für alle Komponenten in einer RTL-Zwischendarstellung (IR), Schätzung der Ankunftszeit des schlimmsten Pfades der RTL IR und schließlich der Aufbau einer konsolidierten blockbasierten Timing-Schätzmaschine, die alle Pins mit Timing-bezogenen Metriken annotiert. Der auf ML-basierende Ablauf liefert hinreichend genaue Timing-Metriken für alle Pins in einer RTL IR, ohne dass eine RTL-Generierung, Synthese oder STA-Läufe erforderlich sind. Darüber hinaus besteht der Hauptvorteil der vorgeschlagenen Methoden darin, dass die Ausführung der ML-Inferenz schneller abläuft als die Ausführung der EDA-Tools.

Die Timing-Schätzungen aus dieser Arbeit identifizieren die kritischen Bereiche des RTL-Entwurfs und leiten Ingenieure dazu an, verschiedene Mikroarchitekturen und Kodierungsstile zu erforschen, ohne dass eine RTL-Generierung erforderlich ist. Darüber hinaus sind Timing-Schätzungen innerhalb von RTL-Generierungs-Frameworks vorteilhaft, bei denen automatische Transformationen Varianten desselben RTL-Entwurfen erzeugen. Anhand dieser Schätzungen kann die beste Variante ausgewählt werden. Eine manuelle oder automatisierte Mikroarchitektursuche könnte zu einer optimalen RTL IR führen, die Timing-Closure schneller erreicht und die Anzahl der Spins während des Designflusses verringert. Somit können frühe ML-basierte Timing-Schätzungen die Markteinführungszeit verkürzen. Alle Methoden in dieser Arbeit wurden in einer industriellen Umgebung bei Infineon Technologies AG entwickelt und evaluiert.

Acknowledgments

The work conducted in this thesis documents research work carried out at Infineon Technologies AG, Germany, in collaboration with the Electronic Design Automation (EDA) Chair at Technische Universität München, Germany.

I want to express my deep gratitude to Hon.-Prof. Dr.-Ing Wolfgang Ecker for his guidance, challenges and learning experiences over these years. Thanks to Dr. Ing. Daniel Müller-Gritschneder for being my mentor and to Prof. Dr.-Ing. Ulf Schlichtmann for reviewing this thesis. I am particularly grateful to Roland Schwenk and Dr. Volkan Esen for allowing me to pursue this doctoral work at the company.

Special thanks are due to my colleagues Endri Kaja and Nicolas Gerlin, whose mutual support was indispensable during these PhD years. I am also thankful to the students who worked alongside me and contributed to this thesis in various ways: Zimin Zhao, Gamze Naz Kiprit, Ishwor Subedi, Prajwal Kashyap, Yuchen Xhi, Giovanni Di Nuzzo, and Mohamed Amine Kthiri.

Finally, I thank my family and friends for their unconditional support. Last but not least, I thank my life partner, Chris, for his support and for bringing joy to my life. Without him, I probably would have finished writing this thesis much faster.

Contents

List of Abbreviations

AI	Artificial Intelligence
AIG	And-Inverter Graph
ALU	Arithmetic Logic Unit
API	Application Program Interface
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
AT	Arrival Time
BAND	Bitwise AND
BFS	Breadth-First Search
BKA	Brent-Kung Adder
BNAND	Bitwise Negated AND
BNOR	Bitwise Negated OR
BOR	Bitwise OR
BXNOR	Bitwise Negated XOR
BXOR	Bitwise XOR
CAGR	Compound Annual Growth Rate
Chisel	Constructing Hardware in a Scala Embedded Language
CIM	Computation-Independent Model
CLB	Configurable Logic Block
CMOS	Complementary Metal–Oxide–Semiconductor
CNN	Convolution Neural Network
ConvGNN	Convolutional Graph Neural Network
CPU	Core Processing Unit
CTS	Clock Tree Synthesis
DAG	Directed Acyclic Graph
DDS	Dynamically Dimensioned Search
DEMUX	Demultiplexer
DGL	Deep Graph Library
DL	Deep Learning
DNN	Deep Neural Network
DRC	Design Rule Check
DSL	Domain-Specific Language
DSP	Digital Signal Processor
DT	Decision Tree

EDA	Electronic Design Automation
EGAT	Edge-Featured Graph Attention Network
EQ	Equal To
FIRRTL	Flexible-Intermediate Representation of RTL
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
GAE	Graph Autoencoder
GBR	Gradient Boost Regressor
GCN	Graph Convolution Network
GDS	Graphic Data System
GDSII	Graphic Data System II
GML	Graph Modeling Language
GNN	Graph Neural Network
GRU	Gated Recurrent Unit
GS	Greedy Stochastic
GT	Greater Than
GTEQ	Greater Than or Equal To
HDL	Hardware Description Language
HLS	High-Level Synthesis
HSI	Hardware Software Interface
HW	Hardware
HWMINUS	Hardware Subtraction
HWMUL	Hardware Multiplication
HWPLUS	Hardware Addition
IoT	Internet of Things
IP	Intellectual Property
IR	Intermediate Representation
ITRS	International Technology Roadmap for Semiconductors
JSON	JavaScript Object Notation
KNN	K-Nearest Neighbor
KSA	Kogge-Stone Adder
LAND	Logical AND
LNAND	Logical Negated AND
LNOR	Logical Negated OR
LOR	Logical OR
LSTM	Long Short-Term Memory
LT	Less Than

LTEQ Less Than or Equal To
LUT Lookup Table
LVS Layout Versus Schematic
LXNOR Logical Negated XOR
LXOR Logical XOR

MAE Mean Absolute Error
MAPE Mean Absolute Percentage Error
MDA Model-Driven Architecture
MDE Model-Driven Engineering
ME Maximum Error
MI Mutual Information
MIG Majority-Inverter Graph
ML Machine Learning
MLP Multi-Layer Perceptron
MoD Model of Design
MoT Model of Things
MoV Model of View
MoV Template of View
MPNN Message Passing Neural Network
MSE Mean Squared Error
MSLE Mean Squared Logarithmic Error
MUX Multiplexer

NEQ Not Equal To
NLDM Non-Linear Delay Model
NLP Natural Language Processing
NN Neural Network

OMG Object Management Group

PDK Process Design Kit
PI Primary Input
PIM Platform-Independent Model
PM Platform Model
PO Primary Output
PPA Power, Performance, and Area
PSM Platform-Specific Model
PVT Process, Voltage, and Temperature

QoR Quality of Result

RAM Random-Access Memory
RAND Reduced AND

RAT	Required Arrival Time
RCA	Ripple Carry Adder
RecGNN	Recurrent Graph Neural Network
ReLU	Rectified Linear Unit
RF	Random Forest
RISC-V	Reduced Instruction Set Computing - Five
RL	Reinforcement Learning
RMSRE	Root Mean Square Relative Error
RNAND	Reduced Negated AND
RNOR	Reduced Negated OR
ROR	Reduced OR
RTL	Register-Transfer Level
RXOR	Reduced XOR
SA	Simulated Annealing
SDC	Synopsys Design Constraints
SGD	Stochastic Gradient Descent
SKA	Sklansky Adder
SoC	System On Chip
SRCC	Spearman's Rank Correlation Coefficient
SSTA	Statistical Static Timing Analysis
STA	Static Timing Analysis
STGNN	Spatial-Temporal Graph Neural Network
SVM	Support Vector Machine
TCL	Tool Command Language
TNS	Total Negative Slack
ToD	Template of Design
TPE	Tree-Structured Parzen Estimator
TPU	Tensor Processing Unit
TSMC	Taiwan Semiconductor Manufacturing Company Limited
UML	Unified Modeling Language
VLSI	Very Large-Scale Integration
WNS	Worst Negative Slack
WS	Worst Slack
XML	Extensible Markup Language

1. Introduction

Trend technologies like the Internet of Things (IoT) and Artificial Intelligence (AI) accelerate the widespread use of Application-Specific Integrated Circuits (ASICs) in industries and consumer electronic products. These chips are designed for particular applications and are valued for their compact size, reduced power consumption, flexibility, and processing performance. On the one hand, ASICs lead to the creation of highly efficient, customized products suitable for a wide range of IoT devices and industrial applications [1]. On the other hand, AI has influenced the semiconductor industry in two distinct ways. Firstly, the rise of generative AI promotes the development of custom chips for cloud services. Secondly, in edge computing, AI workloads are transferred to ASICs for local inference and on-device learning, particularly within the consumer electronics market.

The growth forecasts for these markets are depicted in Figure 1.1, which illustrates the Compound Annual Growth Rate (CAGR) reported by Mordor Intelligence [2]. The CAGR percentage for the IoT, AI chipsets, edge AI, and automotive markets is given from 2024 to 2029. Even though AI chipsets and edge AI hardware will dominate, the ASIC market is also expected by [3] to grow from 18.5 billion to 28.88 billion dollars by 2029. This dynamic landscape pushes semiconductor industries to accelerate the design process and shorten the time-to-market, as they must satisfy the growing demand for application-specific chips in sectors, including Industry 4.0, healthcare, telecommunication, consumer electronics, and automotive. To keep pace with the growing demand for ASICs across various industries, the semiconductor domain continues evolving towards larger-scale and more complex systems. This increasing complexity reflects primarily on the increasing number of transistors or gate counts and shrinking technology nodes. The growth in transistors per chip has been primarily powered by *Moore's law* [4]. Since 1960, this principle states that the transistor density on a chip doubles approximately every two years [5]. However, the doubling rate has slowed due to inevitable physical and

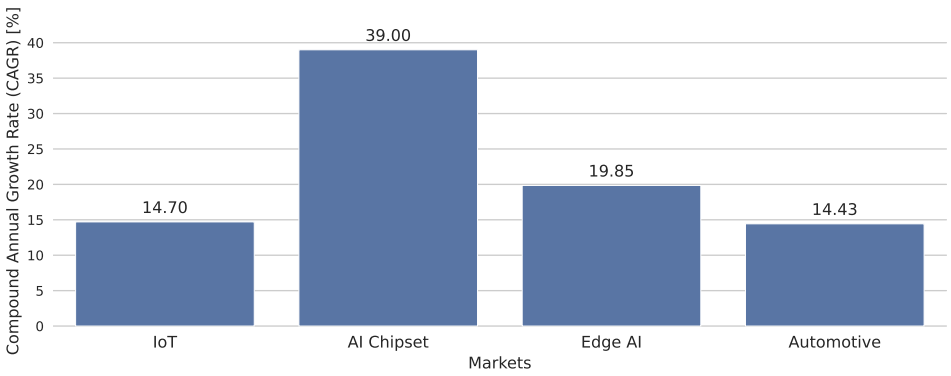


Figure 1.1.: Forecast growth of relevant ASIC-related markets between 2024 and 2029 [2].

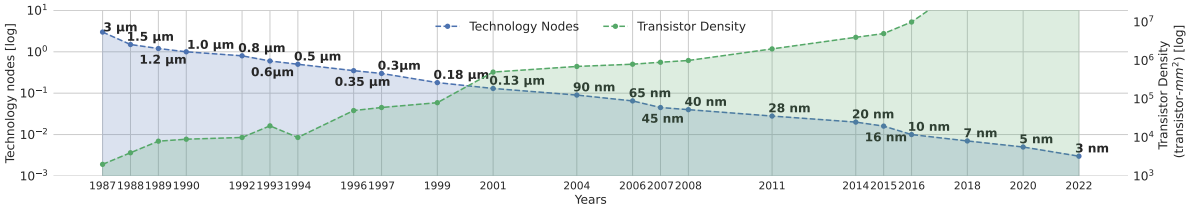


Figure 1.2.: Evolution of commercial technology nodes: Shrinking dimensions [7] and rising transistor density in processors [8].

thermal constraints arising from packing more silicon circuitry into the same compact area. Nonetheless, while the momentum of Moore's law may be slowing, it does not mark the end of advancements in the semiconductor industry. This is evolving beyond the confines of Moore's law following different alternatives, such as the *More-than-Moore* strategy [6].

Despite the challenges, the transistor density per chip continues to rise, and technology nodes persist in their trend of miniaturization, as depicted in Figure 1.2. The blue line in the figure shows the decreasing technology node fabricated by Taiwan Semiconductor Manufacturing Company Limited (TSMC), from 1987 with 3 μm to 2022 with 3 nm [7]. This figure also shows the increasing *transistor density* or transistor count per chip area for Intel processors in the same period [8].

The increasing transistor density and decreasing technology nodes have escalated the design flow's complexity. This has resulted in the so-called *design productivity gap*. This crisis was foreseen in the early 1990s by *SEMATECH* [9]. In 2001, the International Technology Roadmap for Semiconductors (ITRS), a team of international semiconductor experts, also noticed this design gap and stated that the "cost of design is the greatest threat to the continuation of the semiconductor roadmap" [10]. By 2007, the ITRS extended its focus to the design productivity gap, emphasizing the need for a higher level of abstraction to simplify the design process involving hardware and software components [11]. These hardware and software design gaps are depicted in Figure 1.3.

In 2013, the ITRS introduced the *design capability gap*, which represents the discrepancy between the available transistor density and what can be implemented in real-world products. In other words, the circuit capacity following Moore's law pace is outpacing the capability of Electronic Design Automation (EDA) tools to handle this level of complexity effectively [12]. Primary reasons for the design capability gap include a non-agile design flow characterized by repeated iterations or spins, as well as manual and time-consuming tasks. This gap in design capability is the primary motivation behind this thesis, as it aims at accelerating the design flow from the early design stages.

1.1. Problem Description: The Challenges of the ASIC Digital Design Flow

The design capability gap highlights the need for better design flows to cope with the increasing complexity of ASIC designs. Theoretically, the ASIC digital design flow follows a top-down

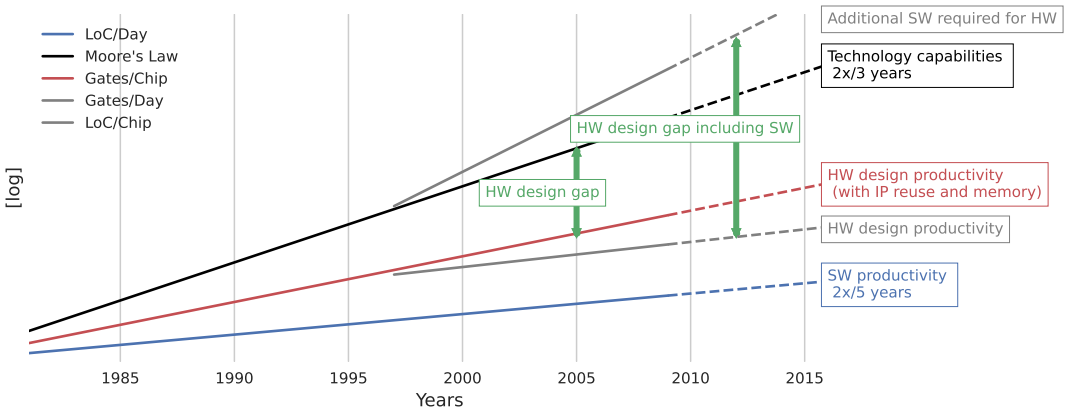


Figure 1.3.: Hardware (HW) design gap influenced by software (SW) productivity measured with lines of code (LoC) as forecasted by the ITRS [11].

approach. In practice, it is a time-consuming, sequential, and iterative process that starts with the chip specification modeling the desired application. Afterward, architecture analysis is conducted to map the target functionality to a collection of interacting modules such as processors, memories, and buses. The behavioral description of those modules is mapped to Register-Transfer Level (RTL) blocks using Hardware Description Languages (HDLs). The transition from system specification to RTL is done manually or by using RTL generation tools or High-Level Synthesis (HLS). The design described in an HDL is the input to the logic synthesis step, which maps the RTL blocks to a combination of cell gates selected from the target technology node. Given the high number of available cells in the technology libraries and the strict design constraints regarding Power, Performance, and Area (PPA), generating the gate-level netlist is complex and iterative. During physical synthesis, the gate-level netlist is mapped to a physical chip layout under several design rules and constraints. The netlist undergoes four main stages: floor and power-planning, placement, Clock Tree Synthesis (CTS), and routing. After functional verification and sign-off checks, the chip layout goes through the manufacturing flow, i.e., fabrication, packaging, and final testing.

In the early 1980s, the field of EDA emerged with a mission to simplify the design flow. Following the *divide and conquer* principle, EDA has incorporated tools to automate each flow stage at the different levels of abstraction, from simulating transistor behavior to logic gates and various types of synthesis to verification. Keeping pace with evolving challenges, EDA has provided automated tools for Very Large-Scale Integration (VLSI), designing chips with billions and trillions of transistors. In addition to the high complexity of the design process, EDA tools are also required to ensure the manufactured chip has no errors or flaws. Otherwise, time and financial resources may be inefficiently used.

Thanks to the efforts of both industry and academia, EDA tools have evolved and allowed the production of today's chips. These tools have significantly improved in four main areas: physical design, simulation, synthesis, and testing [13]. Even though the digital design process is highly automated across different levels of abstraction, the flow encounters significant drawbacks:

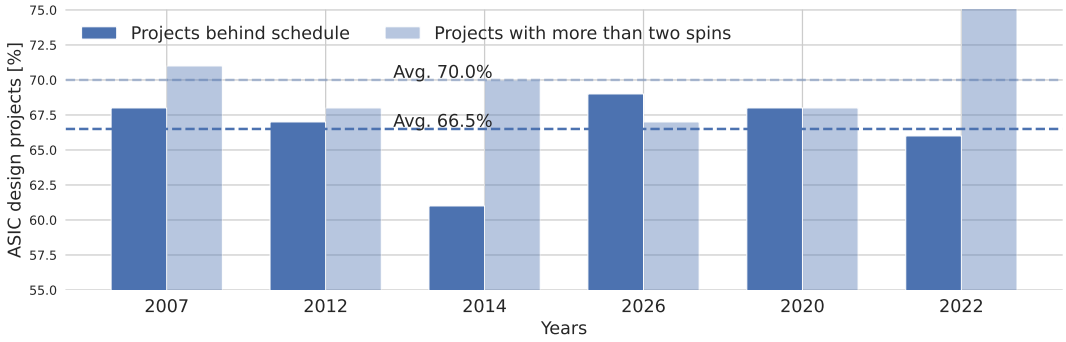


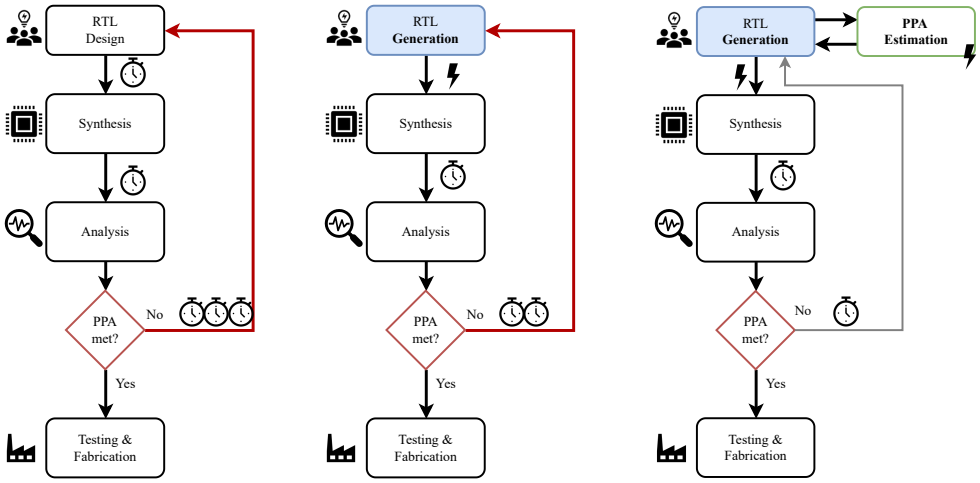
Figure 1.4.: Averaged ASIC projects behind schedule and with more than two spins before production. Based on the data collected by the Wilson Research Group [15]–[18].

1. It relies on the designer's expertise to select the proper configurations of EDA tools.
2. Design space explorations at RTL, logic synthesis, and physical synthesis are manual and, thus, error-prone, limited, and time-consuming.
3. There is no early analysis or predictability of the results.
4. Error detection comes late in the flow. Thus, corrections cause design spins, which are costly in terms of time and resources.
5. EDA tools are standalone and do not work seamlessly across different hardware or software tools, vendors, platforms, or versions.

These limitations underscore the importance of improving EDA tools further to meet the demands of modern chip designs. Along with other factors, these limitations also contribute to the iterative nature of the design flow, which can span days, weeks, or even months. To quantify this, metrics that measure efficiency in ASIC design projects are employed, such as the percentage of projects behind schedule and number of spins required before production. A well-known functional verification study conducted by the Wilson Research Group [14] collects these metrics over different participants worldwide. Although an EDA vendor commissions this study, it has been anonymously conducted from 2007 to 2014 [15], 2016 [16], 2020 [17] and 2022 [18]. Some results are depicted in Figure 1.4. According to the average results from 2007 until 2022, 66.5% of the ASIC projects fell behind their original schedule. Additionally, about 70% of the ASIC projects needed two or more spins before production readiness. Notably, both trends have remained relatively stable despite the increasing design complexity. Nevertheless, there is a clear need for EDA tools to reduce the number of required spins and meet the time-to-market demands. This need is particularly relevant given the expanding ASIC market across emerging sectors such as AI and IoT. Therefore, developing efficient and effective EDA tools that can handle the increasing design complexity is essential.

1.2. Motivation

The market growth of ASICs, whose complexity keeps increasing over the years, has outpaced the advancement of EDA tools, resulting in a challenging design flow that requires several long iterations. All those factors motivate this thesis to accelerate the ASIC digital design flow.



(a) Conventional flow (b) RTL generation (c) PPA estimation

Figure 1.5.: Different strategies of the digital design flow.

A major hurdle of the design flow is the delayed feedback of crucial design metrics. After physical synthesis, timing sign-off checks ensure the final layout meets the initial timing constraints and assess the Quality of Results (QoRs) regarding PPA. In particular, timing violations cause spins of the design flow to implement corrective measures at previous design stages. This process typically involves the analysis of synthesis outcomes, timing reports, and adjustments to the RTL design, as illustrated in Figure 1.5a.

The Wilson Research Group also reports the percentage of flaws by type contributing to spins in ASIC projects [15]–[18]. While these reports encompass a broad spectrum of flaw categories, this thesis focuses mainly on the flaws related to logical or functional errors and timing violations. Figure 1.6 shows the average occurrence of these flaws over the years. Data from 2007 through 2022 reveals that functional or design errors constitute 50.67% of the flaws. Additionally, timing violations, whether they involve paths being either too fast or too slow, account for another 23.33%. It is essential to note that a single ASIC design can be affected by more than one flaw type, each of which may cause a spin of the flow. As a result, the sum of all flaw types per year in Figure 1.6 can exceed 100%.

Notably, the most frequent design functionality or logic flaws arise predominantly from changing, incorrect, or incomplete specifications or errors made during the design process [18]. These flaws are corrected directly in the RTL design. Recognizing the importance of this design phase, RTL generation frameworks are employed to accelerate this design stage by increasing Intellectual Property (IP) reuse and reutilizing existing models, parameters, and generators. Additionally, the concept, design, and verification efforts are also considerably reduced when substituting the manual RTL design process with an automated RTL generation approach, as depicted in Figure 1.5b. Thus, the automation of the initial design stage accelerates the overall digital design process. Nevertheless, the resulting RTL microarchitectures generated by RTL generators may not necessarily consider the design's PPA after synthesis. Instead, they rely heavily on the proficiency of the engineers coding the generators to implement efficient

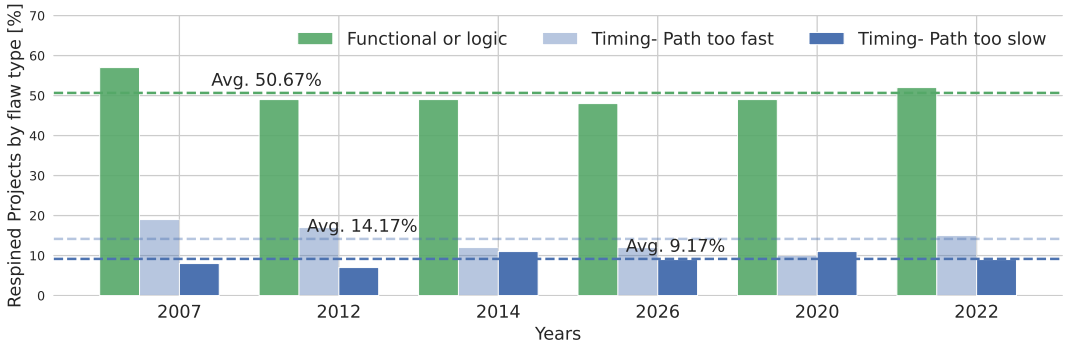


Figure 1.6.: Averaged type of flaws causing spins in ASIC projects. Based on data collected by the Wilson Research Group [15]–[18].

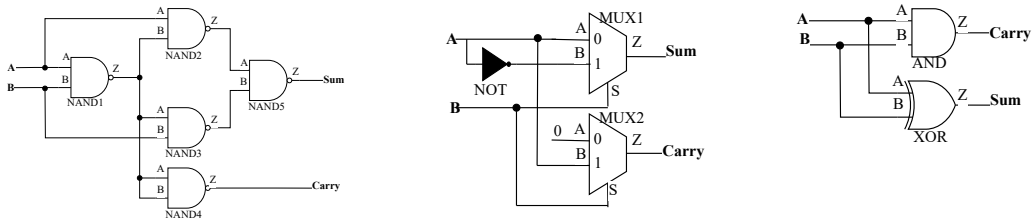


Figure 1.7.: Different microarchitectures for a half-adder.

microarchitectures.

One of the most important aspects of any digital design is achieving timing closure, which ensures the design's correct operation at the designated clock frequency. As depicted in Figure 1.6, a significant 23.33% of the ASIC projects investigated over the years were found to have flaws associated with timing violations. More specifically, these violations were due to the critical path either being too fast or too slow. Unfortunately, information about the critical timing paths of an RTL comes very late, as it implies the execution of all stages in the physical synthesis flow, from floor planning and placement to routing. Each of these stages involves distinct tools and teams, which, unfortunately, results in delayed timing feedback.

In essence, obtaining timing data from an RTL design is complex. However, it is crucial to address timing sign-off violations during the RTL stage [19]. Thus, early timing estimations at RTL are required. Analyzing design metrics earlier in the design flow is known as the *shift-left* methodology. This design paradigm is motivated mainly by the influence of the RTL microarchitecture and HDL coding styles on the PPA results after synthesis [20]. The shift-left methodology aims to identify and fix violations as soon as possible in the design flow. To exemplify this, Figure 1.7 presents three microarchitectures for a half-adder. While each RTL variant delivers the same functionality, they result in different arrival times at the design's primary outputs and, thus, different timing metrics based on the target clock frequency. Without timing estimations, the hardware designer is obliged to choose the best RTL variant based on his expertise and wait until synthesis results from other teams are reported back. The

advantages of incorporating early timing estimations at the RTL stage are manifold. These estimations reduce the *time-to-feedback* and accelerate the design closure and, consequently, the overall design flow. Traditionally, the selection of optimal RTL microarchitectures has been guided by past experiences and expert knowledge. However, with the introduction of timing estimations at the RTL stage, designers can experiment with various RTL variants and HDL styles without needing multiple back-and-forth spins of physical synthesis stages [19]. Similarly, RTL generators rely on the engineer's expertise in coding timing-efficient generators. Hence, RTL generation frameworks also require early feedback regarding the timing behavior of the designs they generate. By integrating timing estimation into an RTL generation framework, as illustrated in Figure 1.5c, we can combine the benefits of RTL generation with early timing feedback, enabling the precise identification of timing bottlenecks in the microarchitecture of a design while maintaining IP reusability, generalizability, and scalability across various designs.

1.2.1. Requirements

Inspired by AI's transformative influence on EDA, this thesis proposes applying Machine Learning (ML) models for gathering early timing estimations. While existing solutions already utilize ML for predicting design metrics, these primarily focus on estimations at HLS or physical design stages. In contrast, this thesis introduces novel methods for estimating timing-related metrics at RTL. State-of-the-art ML techniques are leveraged, as well as, open-source synthesis tools, and the RTL generation framework developed at Infineon Technologies AG. The central research question driving this thesis is: *How can ML be effectively integrated at the RTL stage to obtain early timing estimations and thereby enable optimization?*

Although very powerful, ML is not trivial and faces several challenges. While these challenges are typically common across various fields utilizing ML, this thesis specifically addresses the following issues:

- **Data Availability:** ML models are data-driven methods. Thus, selecting an appropriate data collection flow is crucial. In some stages of the EDA flow, data can be scarce or produce out-of-distribution samples. At RTL, designs can be very complex, and obtaining representative data for training and evaluating ML models can be challenging.
- **Data Annotation:** In a supervised learning setting, each sample of the training dataset needs to be correctly labeled with its timing metrics. If done manually, this can be a time-consuming and error-prone process.
- **Feature Selection:** Identifying the meaningful features that influence the timing behavior of a design at its RTL level is challenging due to the high complexity of a digital design and the timing models of the target technology node.
- **Model Selection and Training:** ML model architectures should cope with the dataset structures. Their hyperparameters and training configuration should be fine-tuned to achieve optimal performance.
- **Fast and Accurate Estimations:** The main benefit of using ML for estimating design metrics is that the inference of ML models is faster than running EDA tools. This must be maintained regardless of the increasing complexity of the designs. ML-based estimations

should be faster than EDA tools but, at the same time, highly correlated with their results.
- **Scalability:** Given the complexity and size of digital designs across different applications, evaluating how pre-trained ML models scale and generalize across small-scale and complex designs is crucial.

Considering the main question and challenges outlined in this thesis, the specific requirements for the proposed ML-based timing estimations are:

$\mathbb{R}$1: A data generation flow shall be established to generate extensive and labeled training designs.
Using an RTL generation framework, training circuits shall be quickly and automatically generated, and open-source tools for ASIC synthesis shall be used to generate timing-related ground-truth labels. This approach allows thousands of runs without incurring license costs and ensures transparency in the synthesis flow.

$\mathbb{R}$2: Features to feed ML models shall provide domain-specific insights.
In the state of the art, ML-enabled timing estimations often use gate-level netlists as inputs. Features such as wire length, parasitic, and wire delays can be easily extracted at that abstraction level. Such features are unavailable in the RTL Intermediate Representation (IR) or RTL designs. Thus, feature analysis shall help select meaningful RTL features that correlate with the timing objectives.

$\mathbb{R}$3: Features to feed ML models shall be extracted from RTL IRs.
While current methods utilize gate-level netlists or Verilog-based RTL to predict post-synthesis timing metrics, this thesis shall extract features from a higher abstraction level, specifically the RTL IR, to eliminate the need for synthesis. The RTL IR level is particularly suitable for feature extraction due to several reasons: (i) It ensures that all designs are platform and HDL-independent, (ii) RTL IRs are prevalent in RTL generation frameworks, facilitating the adaptability of ML models across different design flows, (iii) it enables microarchitecture exploration through model transformations, and (iv) it allows for direct ML inference from RTL IRs, bypassing the need for HDLs generation.

$\mathbb{R}$4: State-of-the-art ML models shall be implemented.
The estimation of timing-related metrics shall be treated as a multi-objective regression task to be solved using shallow or deep ML models. A comparison between architectures shall be performed to identify the most effective approach.

$\mathbb{R}$5: ML-based timing estimations shall be generalizable and scalable.
ML-enabled timing estimations shall be integrated into the RTL generation framework and applied to any generated circuit, regardless of functionality, size, or complexity. To generalize to any circuit, the ML-based flow shall estimate the timing behavior of primitive components, which are the building blocks of more complex designs. The timing estimations shall initially target delays through pin connections inside those primitive components. Unlike state-of-the-art research, the training and evaluation circuits shall not be limited to a few samples.

$\mathbb{R}$6: The timing estimations shall target global timing metrics. Estimating delays of a pin inside a component in the RTL IR assists in detecting critical pins at a component level. Additionally, timing estimations shall leverage pin-to-pin delays to estimate global design metrics, such as slack or arrival times, enabling the comparison of RTL variants or the optimization of RTL IR regions.

$\mathbb{R}$7: The timing estimations shall resemble the Static Timing Analysis (STA) flow. ML models shall implicitly learn the synthesis and STA results of a design and its components. A divide-and-conquer strategy shall be considered as both processes are too complex to be abstracted with a single ML model. A timing estimation engine that follows the classical block-based STA shall be proposed to estimate gate delays, propagate them through a timing path, and calculate timing metrics. ML models shall be integrated into the timing analysis flow, either coarsely or finely, with specific tasks offloaded to the ML models and other tasks solved by classical algorithms for graph traversal, propagation, or minimization.

$\mathbb{R}$8: The timing estimations shall highly correlate with STA tool results yet be faster. Timing estimations shall correlate with the results after synthesis and STA using the same tool configurations employed during training. Additionally, the main benefit of timing estimations is their faster inference time compared to traditional EDA tools.

$\mathbb{R}$9: The timing estimation flow shall be portable to different operating conditions. The timing estimation flow shall be easily portable to different component types, target technology nodes, and timing constraints. As the models' transfer learning capabilities are out of this thesis's scope, the flow shall be easily configured to re-collect training data and re-train the ML models when new component types or technologies need to be supported. In contrast, timing estimation shall support varying timing constraints for clock and primary ports.

$\mathbb{R}$10: Potential use cases of the timing estimations shall be formulated. Timing estimations at the RTL intermediate level shall guide microarchitecture exploration through manual, random, or automated searches, with different microarchitectural transformations implemented at this abstraction level. An initial formulation of these search flows shall be presented.

1.2.2. Envisioned Approach: ML for Early Timing Estimation at the RTL Stage

In the preceding section, several requirements for an ML-based timing estimation flow at the early stages of the design are outlined. This thesis aligns with these requirements and presents an envisioned design flow depicted in Figure 1.8. The flow starts with informal design specifications, which are transformed into an RTL IR using an RTL generation framework. This RTL IR is a technology-independent model representing a design's components and hierarchies. From this tree, features are extracted into tabular or graph datasets. ML models pre-trained using an ML flow are used for three purposes: (i) estimating the delay of input-to-output connections for all components in a design, (ii) estimating its worst-case arrival time, and (iii) building a consolidated block-based STA flow annotating all pins in the design with delays, Arrival Times (ATs), required ATs, and slacks. In cases where the timing constraints are not

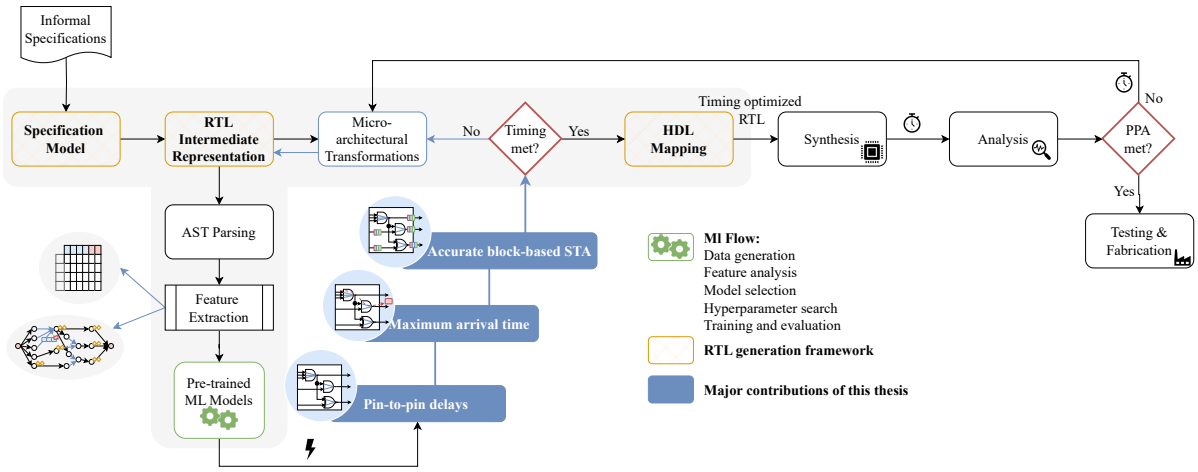


Figure 1.8.: Envisioned digital design flow involving RTL generation and timing estimations.

met, microarchitectural transformations modify the RTL IR, and timing can be re-estimated. Conversely, if the timing constraints are satisfied, the design undergoes synthesis and sign-off analysis. By providing a timing-optimized RTL, the spins of the design flow are reduced. A similar flow for HLS [21] has been concurrently developed with this thesis, demonstrating the validity of the methods in this thesis.

1.3. Thesis Overview

The thesis is organized into eleven chapters:

- Chapter 2 gives an overview of the ASIC digital design. Additionally, it introduces key concepts related to metamodeling, model transformations, and how they have been leveraged to build the in-house RTL generation framework used in this thesis. This chapter concludes with an introduction to the STA of digital circuits.
- Chapter 3 provides a foundation for the ML methods used, especially for solving regression tasks. It details the steps from dataset collection, feature analysis, training, and validation to inference. This chapter also introduces the Graph Neural Network (GNN) deep learning framework and other architectures that were implemented.
- Chapter 4 reviews the state of the art in ML applications for digital design flow tasks. In particular, the gap in applications solving EDA tasks at early design stages is highlighted. It also summarizes both non-ML and ML-based techniques for estimating timing-related metrics. This chapter concludes with a comparison between the reviewed methods and this thesis.
- Chapter 5 focuses on the data generation flow. It details the in-house RTL generation framework used to generate random Verilog-based RTL from informal specifications. It also explains the extraction of ground-truth labels through logic synthesis and STA tools.
- Chapter 6 introduces the use of Deep Neural Network (DNN) to estimate the input-to-output connection delays and slews in components of an RTL IR. Different ML architectures are explored, including shallow ML models, Multi-Layer Perceptron (MLP)

networks, and GNNs.

- Chapter 7 details the use of GNNs to jointly predict pin delays and the worst ATs over all the primary outputs. The latter gives more information on the critical path and enables comparing different RTL variants.
- Chapter 8 presents Fake Timer, an ML-enabled block-based STA flow to obtain accurate timing metrics for all pins and primary ports in the design. This novel method accurately annotates every pin of an RTL IR with relevant timing metrics like ATs, Required Arrival Times (RATs), and slacks. Additionally, this chapter describes the delay correction mechanism added to the Fake Timer to correct the predicted delays and increase the correlation of the computed ATs with flattened and hierarchical synthesis results.
- Chapter 9 describes the experiments conducted and summarizes the results for the pin-to-pin delay estimation, the joint estimation with the worst ATs, and the computed and corrected ATs using the proposed Fake Timer approach.
- Chapter 10 demonstrates the use of timing estimations for microarchitecture search at RTL. It shows microarchitecture transformations that improve the QoRs of the design. Although manual or random search can find optimal microarchitectures, the large design space makes exploration tedious. Finally, this chapter also outlines how this microarchitecture search could be further automated.
- Chapter 11 concludes by summarizing the contributions of this thesis and discussing future research directions in the field.

1.4. Publication List

Parts of this thesis have already been published in the following publications:

[1] D. Sánchez Lopera, L. Servadei, V. P. Kasi, S. Prebeck, and W. Ecker, "RTL delay prediction using neural networks", in *Nordic Circuits and Systems Conference (NorCAS)*, IEEE, 2021, pp. 1–7. DOI: 10.1109/NorCAS53631.2021.9599868.

[2] D. Sánchez Lopera, L. Servadei, G. N. Kiprit, S. Hazra, R. Wille, and W. Ecker, "A survey of graph neural networks for electronic design automation", in *Workshop on Machine Learning for CAD (MLCAD)*, ACM/IEEE, 2021, pp. 1–6. DOI: 10.1109/MLCAD52597.2021.9531070.

[3] C. Lueck, D. Sánchez Lopera, S. Wenzek, and W. Ecker, "Industrial experience with open-source EDA tools", in *Workshop on Machine Learning for CAD (MLCAD)*, ACM/IEEE, 2022, pp. 143–143. DOI: 10.1109/MLCAD55463.2022.9900097.

[4] D. Sánchez Lopera and W. Ecker, "Applying GNNs to timing estimation at RTL", in *International Conference on Computer-Aided Design (ICCAD)*, ACM/IEEE, 2022. DOI: 10.1145/3508352.3561095.

[5] D. Sánchez Lopera, P. Kashyap, N. Gerlin, S. Wenzek, and W. Ecker, "Using open-source EDA tools in an industrial design flow", in *Design and Verification Conference and Exhibition Europe (DVCON)*, 2022, pp. 1–8.

[6] D. Sánchez Lopera, L. Servadei, S. Prebeck, and W. Ecker, "Early RTL delay prediction using neural networks", *Microprocessors and Microsystems (MICPRO)*, vol. 94, 2022. DOI: 10.1016/j.micpro.2022.104671.

[7] E. S. Alcorta, A. Gerstlauer, C. Deng, Q. Sun, Z. Zhang, C. Xu, L. W. Wills, D. Sánchez Lopera, W. Ecker, S. Garg, and J. Hu, "Special session: Machine learning for embedded system design", in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2023, pp. 28–37. DOI: 10.1145/3607888.3608962.

[8] D. Sánchez Lopera, L. Servadei, G. N. Kiprit, R. Wille, and W. Ecker, "A comprehensive survey on electronic design automation and graph neural networks: Theory and applications", *Transactions Design Automation Electronic Systems (TODAES)*, vol. 28, no. 2, 2023. DOI: 10.1145/3543853.

[9] D. Sánchez Lopera, I. Subedi, and W. Ecker, "Using graph neural networks for timing estimations of RTL intermediate representations", in *Workshop on Machine Learning for CAD (MLCAD)*, ACM/IEEE, 2023, pp. 1–6. DOI: 10.1109/MLCAD58807.2023.10299859.

[10] D. Sánchez Lopera, R. N. Kunzelmann, E. Kaja, and W. Ecker, "Fake Timer: An engine for accurate timing estimation in register transfer level designs", in *International Symposium on Quality Electronic Design (ISQED)*, IEEE, 2024, pp. 1–8. DOI: 10.1109/ISQED60706.2024.10528723.

[11] D. Sánchez Lopera and W. Ecker, "AI-enabled static timing analysis at early stages of the digital design flow", in *AI-Enabled Electronic Circuit and System Design: From Ideation to Utilization*, A. Iranmanesh and H. Sayadi, Eds., Springer Nature, 2025.

2. Digital Design Flow

In the earlier days of integrated circuits, the development relied on manual processes, which were not only laborious but also time-consuming. Fortunately, the advancements in technology, computational resources, and programming languages gave rise to EDA. The primary objective of EDA, from its inception to its current state, is to develop tools capable of automatically generating a detailed physical representation of a design based on a desired input behavior and a set of physical rules. As a result, the digital design flow for ASICs follows a top-down methodology, progressing automatically from a higher to a lower level of abstraction, i.e., from the chip concept to the packaged chip, as illustrated in Figure 2.1. Each stage shown in this figure is automated by dedicated EDA tools and flows. Due to the complexity of modern ASIC designs, these stages are distributed among different working teams and tools. Flaws or errors at lower levels of abstraction lead to iterations in the design flow, necessitating back-and-forth communication between teams and tools. Therefore, in practice, the digital design flow is a complex, iterative, and time-consuming process.

The RTL design stage is a prime candidate for increased automation in the early stages of the digital design flow. Traditionally, hardware designers manually implement the target architecture using HDLs, such as Verilog. Hence, the design is error-prone, and its quality depends on the designer's expertise. To automate this stage, RTL generation tools have been proposed to boost IP reuse by increasing flexibility and expedite the translation of high-level descriptions to lower-level RTL code. These frameworks create parameterized and scalable RTL modules, which can then be efficiently integrated into larger designs, thereby improving productivity and design quality. For this reason, Infineon Technologies AG has developed an RTL generation framework to automatically design different Core Processing Unit (CPU) cores and IPs.

An integral aspect of the design process across all stages is timing closure. Both manual and automatically generated RTL designs aim to achieve timing closure at the target clock frequency. To speed up the timing analysis across all components and paths of an RTL design, timing analysis tools are also developed and employed orthogonally to each design stage post-logic synthesis. This thesis explores the application of ML models to the digital design flow, focusing on approximating the timing of RTL designs and leveraging an RTL generation tool. Thus, this chapter initially covers background concepts relevant to the digital design flow and its main stages. In the second part of this chapter, concepts related to Model-Driven Engineering (MDE), metamodeling, and model transformations are introduced, serving as the theoretical foundation for the in-house RTL generation framework employed in this thesis. Finally, this chapter elaborates on the details of the STA, as this thesis aims to approximate such analysis using ML models instead of conventional tools.

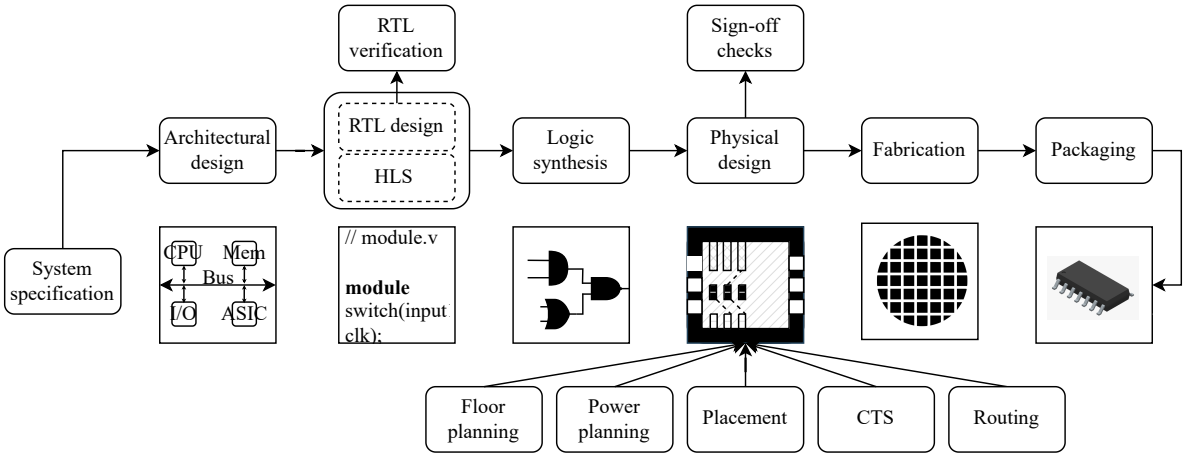


Figure 2.1.: Conventional digital design flow from system specification to packed chips.

2.1. Stages of the Design Flow

As shown in Figure 2.1, the design flow starts with the system specification modeling the desired application. This specification describes the expected functionality, performance, and operating conditions of the design. Afterward, architecture analysis is conducted to map the target functionality to a collection of interacting modules, such as processors, memories, and buses. The behavioral description of those modules is mapped to RTL blocks, such as combinatorial elements implementing logic Boolean functions or storage elements such as flip-flops [22]. The RTL design can be synchronous if there is at least one flip-flop and if the flip-flops are connected to a common clock signal or clock signals with known relationship. Otherwise, the design is purely combinatorial. Hardware engineers describe these blocks at RTL using HDLs such as Verilog and SystemVerilog. The transition from system specification to RTL can be done in different ways, using HLS or, as in this work, using RTL generation frameworks. For instance, HLS automatically converts C or System-C code to Verilog. After the RTL is designed, verification flows check the resulting design against the initial specifications.

Afterward, logic synthesis converts an HDL design into a technology-mapped gate-level netlist. The required inputs to the logic synthesis tools are the HDL-based RTL code and a technology library file [23] available in a Process Design Kit (PDK). The latter is a set of data files describing the physics below the gate-level representation of a chip for the EDA design tools. A PDK developed by a foundry contains technology details, design rules, standard cell libraries, and layout information [24]. The logic synthesis process is composed of three main steps. First, syntax analysis is done to detect syntax errors in the HDL program according to the language rules. Second, the RTL is compiled into a Boolean data structure, and Boolean minimization is performed. HDL constructs are mapped to hardware components. At the end of this elaboration step, the design is mapped to generic technology-independent logic gates. Finally, the components are mapped to standard cells available in the technology library, and the netlist undergoes several iterations of logic and design optimization to minimize gate count, area, and timing [22].

In physical design, five main steps are executed: *floor and power planning*, *placement*, *clock insertion*, and *routing*. The floor plan is the mapping from the gate-level netlist to an initial physical description of the design [25]. During the floor planning stage, relevant features of the physical layout, such as the die area and utilization ratio, are defined. Furthermore, input-output pins and macros are placed in the chip. Macros are major, pre-designed, and functional blocks of an RTL design, e.g., blocks of Random-Access Memory (RAM), which are not mapped to standard cells but must be placed entirely in a portion of the chip. During power planning, the type of power distribution and the position of power pads are decided, and a power grid is created. A good floor plan helps to reduce routing and congestion issues and to quickly meet timing and area targets in subsequent flow stages [25].

During placement, physical realizations of standard cells of the netlist are given an exact location in the chip while optimizing PPA metrics. The placement step is divided into *global* and *detailed placement*. In the former, the standard cells are coarsely divided into groups, and the number of connections between such groups is minimized. In detailed placement, a legal placement for each cell is decided while minimizing wire length and optimizing timing and area [26].

CTS is defined as the routing of the clock signal from its source to all the sequential elements in the chip. This clock sharing is done with the aim of minimizing clock *skew*. The latter refers to the variation of the arrival times of the clock signal at the different chip regions. The clock signal is distributed across the chip using a network of buffers, inverters, and wires in a tree-like structure. This clock tree or distribution network significantly affects power consumption, timing, area, and signal integrity [27].

After the distribution of the clock signal, routing is conducted to connect other signals and blocks. Routing is the most critical stage in the design flow, as patterns of wires and layers should be found to connect all placed cells and macros. It involves determining the optimal paths for the wires while minimizing the wire length, maintaining signal integrity, and reducing power consumption. The routing stage is divided into two steps: *global* and *detailed* routing. In the former, the routing regions are divided into many global routing cells, and fast grid routing is conducted to minimize the wire length and congestion. In the detailed routing step, the global route plan is used to assign specific tracks to each net, create wires, and connect pins of all cells to their nets. Optimal routing paths have a trade-off between congestion and timing results [28].

Once the routing stage is complete, sign-off checks are performed to verify that the design meets the required functionality and constraints before it can be sent for tape-out or fabrication. Sign-off checks include timing and power analysis, Design Rule Check (DRC), and Layout versus Schematic (LVS) [29] comparison. Timing analysis verifies that the design works at the target clock frequency. This analysis is the main focus of the thesis and will be explained in more detail in Section 2.3. Power analysis estimates the design power consumption in idle and dynamic stages. DRC determines if the generated layout of a chip satisfies design rules such as metal width, pitch, and spacing requirements for different layers. Design rules restrict a design's geometry and connectivity to ensure sufficient margin for variability in the manufacturing process [29]. Lastly, LVS verifies if the gate-level netlist and the generated layout

are functionally equivalent. Finally, once these checks are performed, all the intermediate files are combined to generate the final Graphic Data System II (GDSII) file, describing the complete layout.

The sign-off-ready design is sent for manufacturing. The fabrication process involves photolithography, etching, and deposition. First, a photomask is produced to transfer the GDSII design to the silicon wafer. Chemical etching removes undesired material from the wafer and leaves the design patterns. A metal or insulator is deposited on the wafer to create the interconnects. Finally, the chips are separated from the wafer and individually packed. Further testing is required before integrating ASICs into larger, more complex systems.

While this thesis primarily concentrates on the early stages of design, the previous review of the entire flow provides valuable insights into the various levels of abstractions, the immense complexity of the design process, and the necessity of producing an optimized "right-at-first" RTL design.

2.2. An Automated Flow from Specifications to RTL Designs

As depicted in Figure 2.1, the first stages of the design flow go from design specifications to RTL implementations. As already mentioned, the translation from specifications to HDL can be done using HLS or RTL generation tools [30]–[32]. The latter has emerged thanks to the development of programming languages like Python and Scala. These frameworks offer to build highly customizable circuit generators by taking advantage of new programming paradigms, such as object-oriented and functional programming. Specifically, the framework called *Constructing Hardware in a Scala Embedded Language (Chisel)* [30] uses Scala as a Domain-Specific Language (DSL) to describe digital designs at the RTL. Chisel compiles Scala descriptions using the Flexible-Intermediate Representation of RTL (FIRRTL) and generates the output Verilog code, which can be synthesized and simulated. Similarly, *PyVerilog* [31] leverages Python for Verilog generation and analysis. PyVerilog provides four main blocks: a parser transforming Verilog code to an Abstract Syntax Tree (AST), data and control flow analyzers, and a Verilog generator mapping the PyVerilog AST to Verilog. Lastly, *MetaRTL* [32] applies the Model-Driven Architecture (MDA) [33] approach to RTL generation using Python. MetaRTL uses templates, models, and metamodels to construct customizable generators that, starting from specification, produce Verilog-based RTL.

This thesis employs MetaRTL, the Infineon in-house RTL generation tool developed using metamodeling concepts, including MDE and MDA. Thus, the subsequent sections provide a concise overview of these concepts and culminate with an introduction to MetaRTL, detailing its primary views of a design and possible model transformations.

2.2.1. Model-Driven Engineering (MDE)

With the increasing complexity of software systems and the growing demand for code reusability, various methodologies have been explored, with a particular focus on representing software

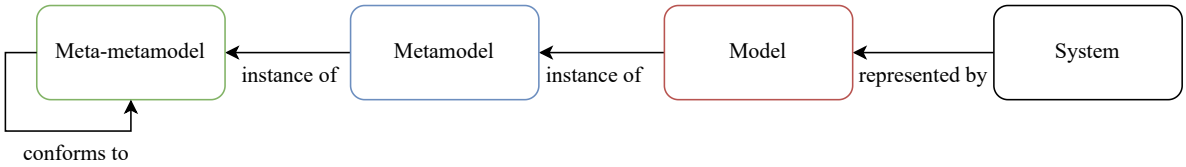


Figure 2.2.: Basic MDE blocks from the system level (rightmost block) to the meta-metamodel (leftmost block) [36].

components as *models*. Models are high-level abstractions that capture the structure, behavior, and requirements of a system within a specific domain. This abstraction brings simplicity and enables the handling of the domain complexity. In this context, MDE has emerged as a software development practice that leverages domain-specific models for the analysis, simulation, and generation of complex systems. MDE involves the abstraction of a system using models, along with the utilization of *metamodels*, *transformations*, and target code generation. Through systematic transformations, abstract models are progressively refined into more concrete representations, ultimately leading to the generation of target code [34].

2.2.1.1. Metamodeling

One of the critical aspects of MDE is the use of metamodels or "models of models". Ecker and Schreiner [35] define models as a description of an entity using concrete attributes, properties, and sub-entities. In addition of models being composed of sub-models, "metamodels identify involved entities and define their attributes and relations" [35]. In this way, the complete structure of a system can be described in terms of models, their relationships, and constraints. While a metamodel describes the possible interconnection of elements, a single model is just an instance of the metamodel with specific attributes, i.e., a model conforms to its metamodel. Moreover, a metamodel is another abstraction of the model, emphasizing the properties of the model itself. Additional levels of abstraction can be created by appending the word "meta". In this sense, a meta-metamodel is a model of a metamodel [35].

Figure 2.2 shows the different models and their hierarchical relationships. The system represents the view of the "real-world" application to be abstracted. The model is the first level of abstraction and is an instance of the metamodel. Similarly, the structure of the metamodel is given by the meta-metamodel, which defines its elements and constraints. The process of creating models of models is called *metamodeling*. The latter involves defining a domain-specific language and rules to transform models across the different abstraction layers [35].

2.2.1.2. Model Transformations

In the context of MDE, transformations serve as automated mechanisms to convert a model to another at the same or different abstraction level but of the same system. Model transformations find several applications in MDE, including the generation of lower-level models and target code from high-level abstractions, synchronization of models at the same abstraction

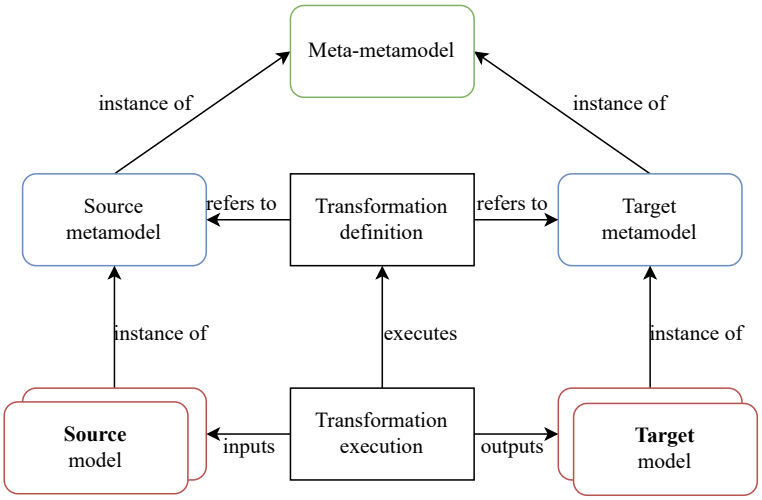


Figure 2.3.: Basic blocks of a model-to-model transformation in MDE [37].

level, and the reverse engineering of high-level models from lower abstractions [37]. As outlined in [37], model transformations are commonly classified into three major categories: *model-to-text*, *text-to-model*, and *model-to-model*. The former transformations output texts or strings and are helpful in generating artifacts, such as code or documentation. The *text-to-model* transformations consist of traditional parsing approaches. The latter transformations generate an instance of the target metamodel and are the most commonly used. Different approaches, such as direct manipulation, structure-driven, template-based, and graph-transformation-based, are employed to implement transformations in an MDE framework [37].

Figure 2.3 shows the basic flow of model-to-model transformation as explained in [37]. The transformation converts one (or several) input model(s) to one (or several) target model(s). The transformation is defined at the metamodel level but executed on the instantiated models. Moreover, both metamodels for source and target conform to the same meta-metamodel.

2.2.2. Model-Driven Architecture (MDA)

In 2000, the Object Management Group (OMG) [38] announced a methodology shift from an object to a model-driven architecture. In other words, they went from "everything is an object" to "everything is a model" [39]. MDA is a software standard that supports MDE. Specifically, MDA standardizes software development using metamodeling, model-to-model transformations, and code generation [35]. In contrast to MDE, MDA's scope is reduced, covering only the architectural definition of a software system but leaving the process, how to produce and coordinate models, and their analysis undefined [34].

According to the MDA standard, developing software implies the definitions of models conforming to different metamodels. In particular, MDA proposes an architecture of four models at different levels of abstraction and a set of transformations to get a final system implementation. These are shown in Figure 2.4. First, the *Computation-Independent Model (CIM)* or

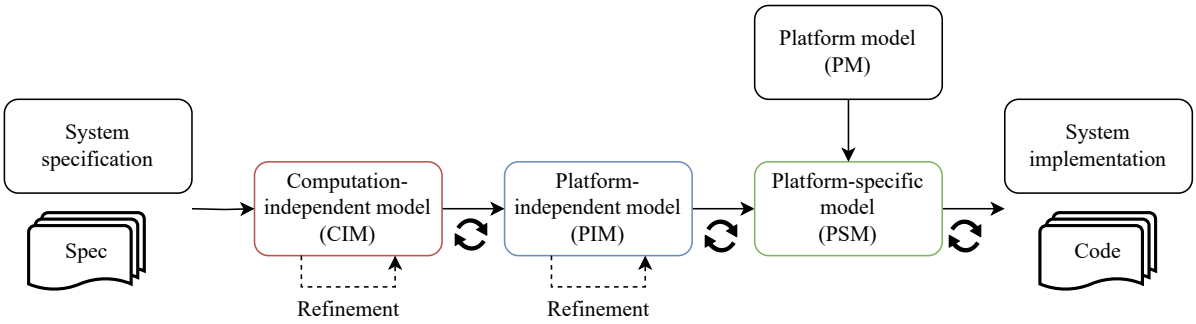


Figure 2.4.: Basic blocks of MDA and possible model-to-model transformations.

domain model describes the system, its requirements, and what it is expected to do. The CIM does not provide implementation or technical details. The *Platform-Independent Model (PIM)* focuses on the system's operation. It describes the system's behavior and functionalities without being specific to any platform. It is the intermediate representation bridging the gap between the domain model and the target implementation. PIMs reveal features that are common among different platforms. Finally, the *Platform-Specific Model (PSM)* combines the PIM with additional features specific to a platform or *Platform Model (PM)*. The PSM describes the computation flow on a specific platform. The latter refers to the underlying infrastructure where a target code implementation is executed, referring to hardware architectures and software components to run on top, such as kernel libraries [40].

Those models are passed through a sequence of model-to-model transformations until the final code is obtained. As defined in MDE, model-to-model transformations map a source input to a target output model. Based on this, the model transformations that can occur in MDA go from a high to a lower abstraction level, i.e., from CIM to PIM, PIM to PSM, and PSM to code. Moreover, transformations refining a model, i.e., in which input and output are at the same abstraction level, are also possible for CIMs and PIMs. On the other hand, transformations from PSM to PIM, PIM to CIM, and PSM to CIM are not feasible [40].

An increasing number of standards are defined to represent domain-specific models, metamodels, and model transformations to promote the MDA in software development. Some standard modeling languages including the Unified Modeling Language (UML) [41]- and its extensions-, are used to mainly describe two levels of models, the PIM or one or more PSMs. The Extensible Markup Language (XML) [42] is mostly employed for the interchange of models [33]. Even though MDA and its drawbacks regarding MDE have been discussed in the literature [39], it has been used in many fields for the entire development cycle of software systems. Examples are the UML-based web engineering [40] and the Infineon in-house RTL generation framework, MetaRTL.

2.2.3. MetaRTL: MDA Applied to RTL Generation

MetaRTL is a generation framework that leverages MDA concepts and standards for the generation of HDLs to describe designs at RTL. In other words, MetaRTL uses models, metamodels,

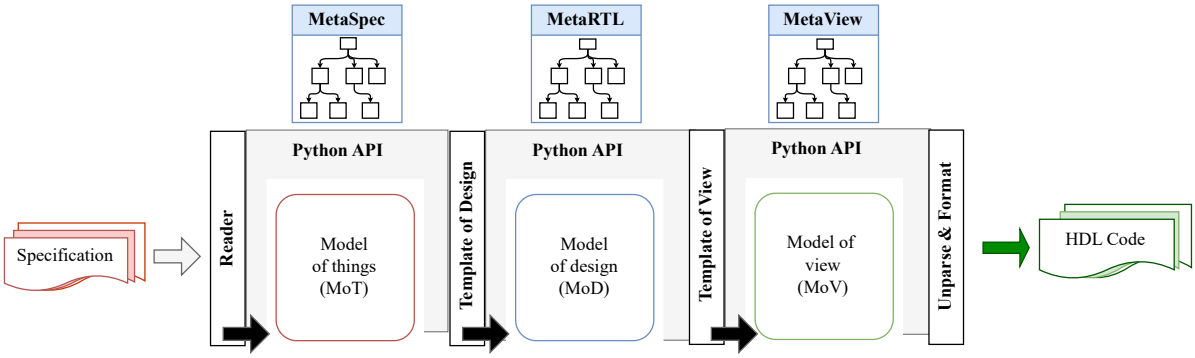


Figure 2.5.: MetaRTL building blocks [32].

and model-to-model transformations to go from design specifications to Verilog implementations [32].

Inspired by MDA, MetaRTL uses three models, as shown in Figure 2.5: the initial specifications or *Model of Things (MoT)*, the design microarchitecture or *Model of Design (MoD)*, and finally, the mapping to an HDL or *Model of View (MoV)*. Each of these models is an instance of different metamodels, such as *MetaSpec*, *MetaRTL*, and *MetaView*, respectively. The generation process starts by reading the informal specifications and translating them into an MoT. The metamodel defines specification components, attributes, and relations, while the MoT is an instance with determined attributes. Template-based transformations convert the MoT into an MoD. The *Template of Design (ToD)* is a DSL implemented in Python. The MoD is a particular instance of a digital design conforming to the MetaRTL metamodel. As this intermediate model is particularly relevant in the thesis, more details are presented in the next section. Additional template-based transformations are then applied to convert the MoD to an MoV. *Template of Views (MoVs)* provide a specific mapping from intermediate models to the target code. The latter is a specific HDL with defined semantics and syntax. Platform-specific details are also considered before the final HDL code or views are generated.

The goal is to keep HDL-specific features closer to the final implementation and further from the initial specification models. In this way, more parameterization and optimization at intermediate stages are possible without considering the complexity of the different target HDLs. Adding different views implies generating proper PMs and MoVs without modifying the models at higher abstraction levels. MetaRTL has been used to develop Reduced Instruction Set Computing - Five (RISC-V)-based CPU cores [43], AI lightweight accelerators [44], and simple IPs such as the register file. The following section showcases the register file to demonstrate how MetaRTL can be used to go from informal specifications to RTL.

2.2.3.1. A Running Example: The Register File

Registers are storage locations inside a CPU design. A register file refers to an array of general-purpose registers utilized for reading and writing data in accordance with an instruction set architecture. A register file can be implemented as an array of flip-flops or a static RAM. While

RAM offers faster performance, its synthesis for Field Programmable Gate Array (FPGA) or ASIC platforms can pose challenges in the absence of specific definitions for the RAM IP. The informal and general specifications of a register file generator are:

1. The number of writing and reading ports is specified
2. The number of registers or flip-flops used in the register file is specified.
3. A register is defined by its size and has a reset value. The former specifies the number of bits that a register stores. The latter is the value the register takes after an asynchronous reset.
4. All interface signals are synchronous to the clock.
5. An asynchronous reset resets the registers to a reset value.

Given these specifications, the design generation flow is summarized in Figure 2.6. Using UML, the meta-metamodel is specified to describe the main components of this IP. In the meta-metamodel, only the main components and attributes of the meta-model are modeled. The MoT specifies one instance of the meta-metamodel, defining the type of register file and the number of read/write ports and registers. The functionality and interconnection of those components are specified in the ToD using Python. The output of this stage is the MoD serialized as an XML tree, which describes the circuit microarchitecture. The MoD implements the design using two multiplexers, one demultiplexer, and a set of flip-flops primitives. The MoD is transformed into Verilog files using pre-defined template-based transformations. The implementation of these transformations is a one-time effort process.

2.2.3.2. MoD or RTL Intermediate Representation

Analog to the PIM layer in the MDA flow, the MoD describes the functionality of a specification model. In MetaRTL, the MoD represents the bridge between the informal specifications and the HDL-specific implementation. The MoD is the microarchitecture of the design describing its structure and behavior. Thus, this intermediate layer gains particular importance as it allows the abstraction of any digital design as an AST of a design's submodules, components, ports, constants, and connections. Figure 2.7 shows the tree structure of an MoD or RTL IR. The MoD considers the design hierarchies using component and *connection* objects. These link the ports and constants on the same hierarchy and interconnect the different levels. Throughout this thesis, the terms MoD and RTL IR are used interchangeably.

Components of an MoD or RTL IR can be descriptive, behavioral, sequential as flip-flops, and *primitive* or basic. Mainly, the primitive components are the fundamental logic functions of digital designs, including operators such as AND, OR, NOT, and XOR in negated, reduced, logical, and bitwise fashion, arithmetic operators for addition, subtraction, and multiplication, multiplexers and demultiplexers, and comparators. Table 2.1 lists the primitive components considered in this thesis with their minimum and maximum number of possible inputs.

Except for the demultiplexers, all components have one single output. The bit width of this output depends on the type of operator applied over the input operands. The NOT operator, for instance, inverts the input, resulting in an output size equal to the input bit width. Logic

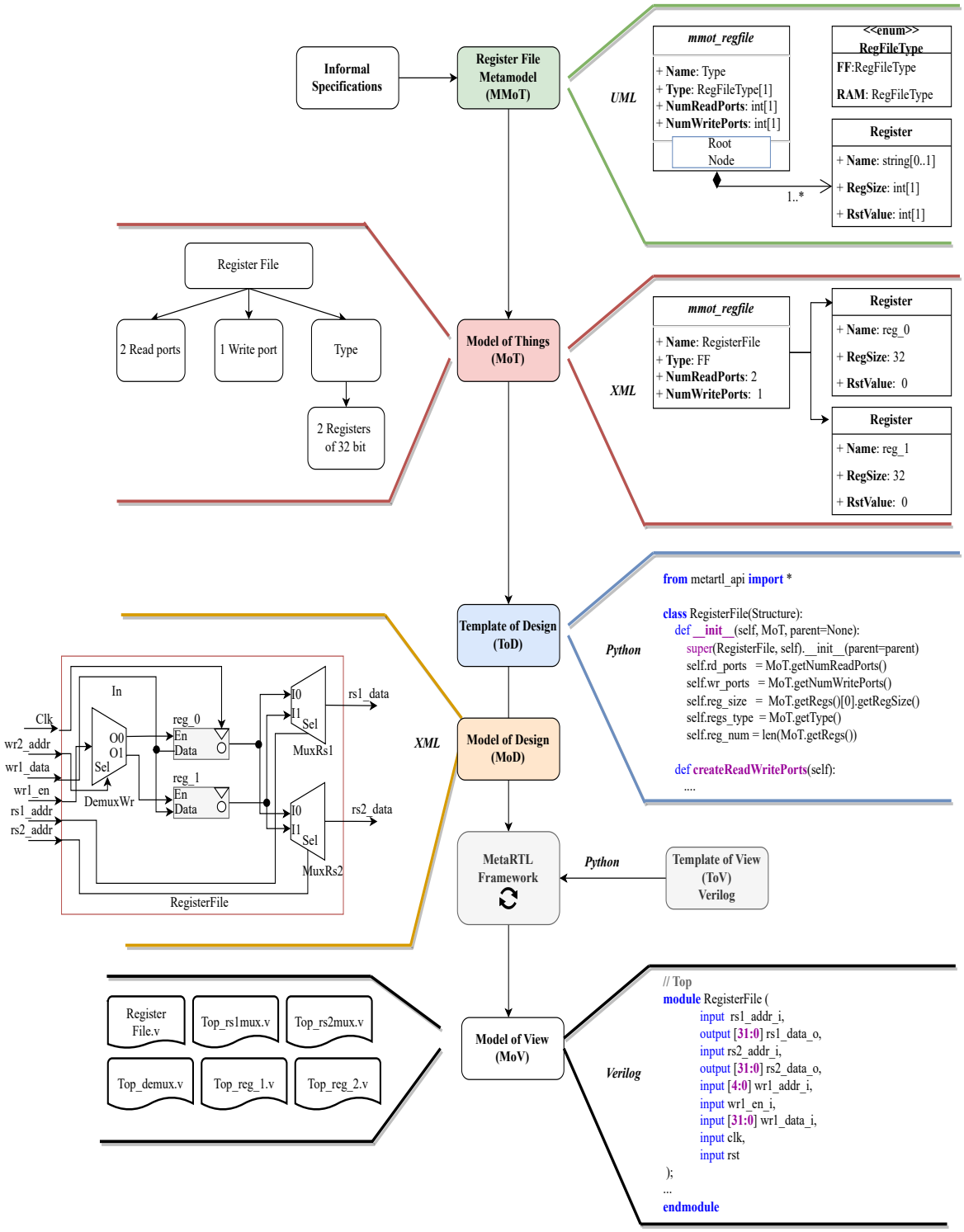


Figure 2.6.: Register file generation flow using MetaRTL.

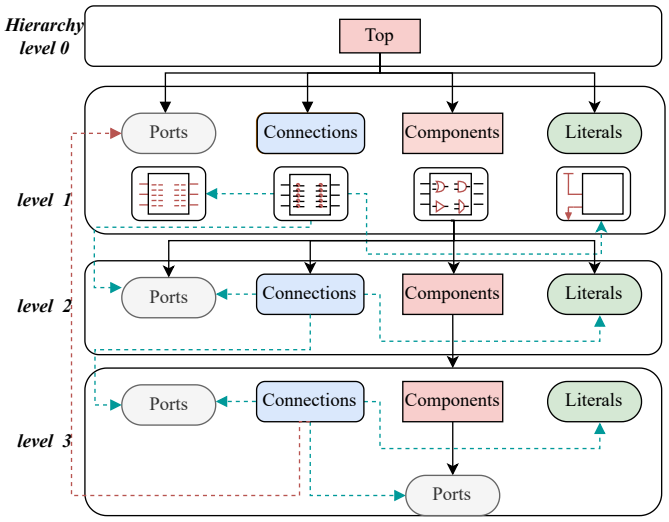


Figure 2.7.: MoD or RTL IR structure used in MetaRTL [32].

Table 2.1.: Basic components used to build an MoD and number of input ports.

Component Category	Full Name	No. Min. Inputs	No. Max. Inputs
Logic	NOT	1	1
	Reduced AND (RAND), Reduced Negated AND (RNAND), Reduced OR (ROR), Reduced Negated OR (RNOR), Reduced XOR (RXOR)	1	*N*
	Logical AND (LAND), Logical Negated AND (LNAND), Logical OR (LOR), Logical Negated OR (LNOR), Logical XOR (LXOR), Logical Negated XOR (LXNOR)	1	*N*
	Bitwise AND (BAND), Bitwise Negated AND (BNAND), Bitwise OR (BOR), Bitwise Negated OR (BNOR), Bitwise XOR (BXOR), Bitwise Negated XOR (BXNOR)	2	*N*
Arithmetic	Hardware Addition (HWPLUS)	2	N
	Hardware Subtraction (HWMINUS), Hardware Multiplication (HWMUL)	2	2
Comparator	Equal to (EQ), Not Equal to (NEQ), Greater than (GT), Greater than or Equal to (GTEQ), Less than (LT), Less than or Equal to (LTEQ).	2	2
Selectors	Multiplexer (MUX)	2	*N*+1
	Demultiplexer (DEMUX)	2	2

operators are further categorized into *reduced*, *logical*, and *bitwise* operators. The reduced variant combines the inputs into a new vector and applies the logic operation bit by bit, yielding one output with a width of one bit. Logical operators align with their definitions in programming languages, reducing operands of higher bit widths first before applying a bitwise operator, also resulting in a one-bit wide output. Conversely, bitwise operators execute the operation elementwise or bit by bit, with the output size corresponding to the maximum bit-width among the operands.

Arithmetic operators assume signed operands to be encoded using the two's complement. Operators such as HWPLUS and HWMINUS provide an output carry as well as an overflow bit. Thus, the bit width of the output is the maximum bit width among the operands plus one carry and one overflow bit. For HWMUL, the bit width of the output corresponds to the sum of the bit widths among the operands. Comparators, designed to compare two input values, feature a single output port with a bit width of one, and the operator's name denotes the type of comparison performed. The size of multiplexers and demultiplexers depends on the size of the input operands. Table 2.2 provides the Verilog syntax of the mentioned components. The negated components are not listed, as they can be derived using the operator $\sim$ in front of the Verilog expressions.

Table 2.2.: Basic components of an MoD using MetaRTL and Verilog syntax.

Component Type	**MetaRTL Expression**	**Verilog Expression**
MUX	MUX(a, b, s)	$s == 1'b1?\ c = a\ :\ c = b$
DEMUX	DEMUX(a, sel)	$s == 1'b1?\ c = a\ :\ d = a$
RAND	RAND(a)	$\&a$
ROR	ROR(a)	$\|a$
RXOR	RXOR(a)	$\wedge a$
LAND	LAND(a, b)	$(\|a)\&(\|b)$
LOR	LOR(a, b)	$(\|a)\|(\|b)$
LXOR	LXOR(a, b)	$(\|a) \wedge (\|b)$
BAND	BAND(a, b)	$a\&b$
BOR	BOR(a, b)	$a\|b$
BXOR	BXOR(a, b)	$a \wedge b$

Component Type	**MetaRTL Expression**	**Verilog Expression**
NOT	NOT(a)	$\sim a$
HWPLUS	HWPLUS(a, b)	$a + b$
HWMINUS	HWMINUS(a, b)	$a - b$
HWMUL	HWMUL(a, b)	$a * b$
EQ	EQ(a, b)	$a == b?$ 1'b1 : 1'b0
NEQ	NEQ(a, b)	$a! = b?$ 1'b1 : 1'b0
GT	GT(a, b)	$a > b?$ 1'b1 : 1'b0
GTEQ	GTEQ(a, b)	$a \geq b?$ 1'b1 : 1'b0
LT	LT(a, b)	$a < b?$ 1'b1 : 1'b0
LTEQ	LTEQ(a, b)	$a \leq b?$ 1'b1 : 1'b0

2.2.3.3. MetaRTL Transformations

MetaRTL defines *transformations* as generators transforming or refining MoDs while maintaining the design functionality. Without transformations, required changes to a design would need to be done in the ToD and the MoD would need to be re-generated. The flow in Figure 2.5 is modified in the intermediate layer by adding Python transformations using the MetaRTL Application Program Interface (API). Figure 2.8 shows the transformation concept on MetaRTL, where an MoD is refined to obtain an output MoD. This refinement targets specific design features without modifying models or templates at other levels of abstraction.

MetaRTL considers four types of transformations. First, *basic* transformations allow the search and identification of components, ports, and connections across the design. They provide graph-based operators such as adding, deleting, and replacing. These core operators locate a

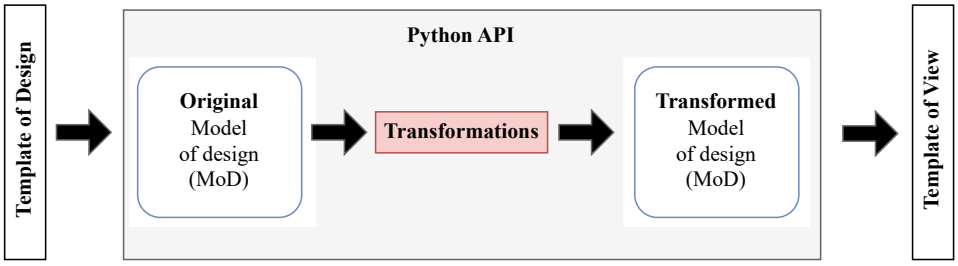


Figure 2.8.: Transformation concept in MetaRTL.

primitive component in an MoD, and replace it with a new equivalent, or delete it completely. In any case, the transformations take care of maintaining valid connections and interfaces. *Generic* transformations convert not only specific small components of the design but the complete design. Examples of generic transformations involve flattening models, updating objetc names and reduction of unused ports. Transformations adding new modules to complete System on Chips (SoCs) can be *design-dependent* or *independent*. Scripts must be rewritten for each target design in design-dependent transformations, such as adding a debugger JTAG module [45]. In contrast, design-independent transformations are generalizable to any SoC, such as adding safety mechanisms via selective hardening [46].

The main benefit of MoD transformations is the reduction of design efforts, as a single transformation can be applied to any input MoD without requiring manual modifications in the ToD for updating ports, structures, or connections [45]. Moreover, on an existing MoD, multiple transformations can be applied to generate different variants of MoDs and different HDL mappings. Transformations facilitate the automated and efficient modification of microarchitecture details within an RTL IR. Thus, they enable microarchitecture variant building and trade-off analysis at the early stages of the design flow.

Figure 2.6 shows the flexibility of MetaRTL by modeling attributes such as the type of register file, using RAM or flip-flops, and, in the latter case, the number and size of registers. MetaRTL transformations could take any MoD of an SoC and replace its register file module with one with different specifications, all without necessitating the regeneration of the whole SoC design. Equivalence checks are performed between the Verilog generated using the original and the transformed MoDs to validate the preservation of the design's functionality and interfaces.

2.3. Static Timing Analysis

In the digital design flow shown in Figure 2.1, timing analysis determines whether a design operates at the required clock frequency and can be performed at any stage of the flow after logic synthesis. However, its results become more accurate after routing, as it considers parasitic and wire delays, which affect the timing behavior of the design. Timing analysis can be *dynamic*, using simulation over different input vectors, or *static*, where no input vectors are required to simulate the complete circuit. Instead, *STA* propagates gate and wire delays from the primary inputs to the design endpoints, such as input to sequential elements or primary output ports. STA covers all paths in a design. Thus, it is more efficient and faster than dynamic analysis [47].

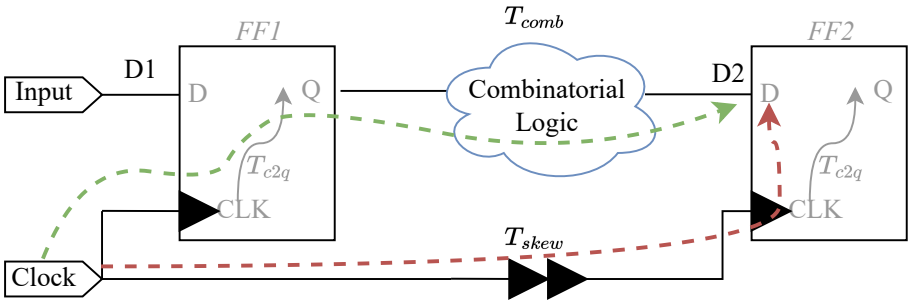


Figure 2.9.: Basic synchronous circuit.

However, it can also waste runtime analyzing false timing paths, i.e., paths never executed in the design.

The accumulated delay at the designs' endpoints is the AT for that given path. In synchronous circuits like shown in Figure 2.9, the AT represents the time the data arrives at the input of the receiving sequential element or flip-flop. It can also be defined as the time required for the input to travel through the data path, represented as a dashed green line. For instance, the AT at the input *D2* of the flip-flop *FF2* is described in Equation 2.1,

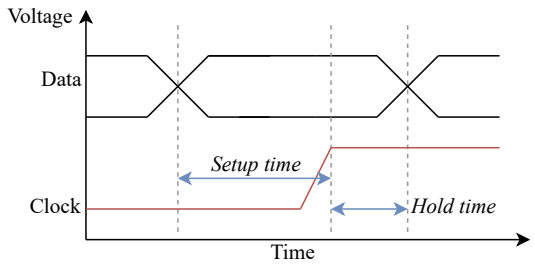
$$AT = T_{c2q} + T_{comb} + T_{c2d}, \tag{2.1}$$

where T_{c2q} is the time from the clock to the output pin of the flip-flop *FF1*, T_{comb} is the time needed for the combinatorial logic between the flip-flops, and T_{c2d} is the time from the clock signal to the input pin *D2*. The RAT is when the data must be at a design's endpoint to ensure the correct functionality of the whole design. The RAT is also defined as the time the clock signal travels through the clock path, highlighted as a dashed red line in Figure 2.9. Thus, the RAT is determined based on the clock period and timing constraints. The RAT at the input *D2* of the flip-flop *FF2* is described in Equation 2.2,

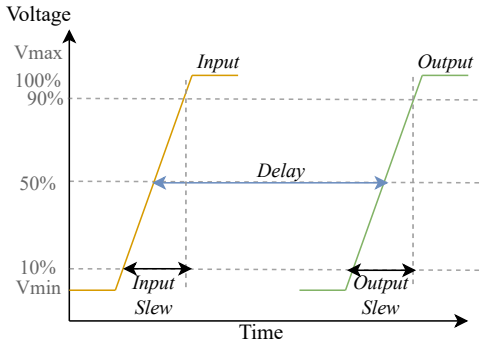
$$RAT = T_{clk} + T_{cp} - T_{skew} - T_{setup}, \tag{2.2}$$

where T_{clk} is the target clock period, T_{cp} is the clock path delay to *FF2*, T_{skew} is the uncertainty or variability of the clock circulating via a clock tree, and T_{setup} is the time required for the data to be stable at *FF2* before the clock edge.

The difference between the actual RAT and AT is a severity measure called *slack*. A positive slack means that gates in the path can be replaced by smaller but slower ones. In this way, the design area can be optimized without incurring timing violations. A negative slack indicates that the signal through the analyzed path is late, and its gates should be optimized or replaced by faster ones with potentially higher areas. The path with the worst slack is called the *critical path* of the circuit [48]. STA tools use the PERT method [49] or others to traverse the timing graph, propagate slews, and find critical paths of the design. The resulting *timing reports* give information about a design path, its start and endpoints, gates, and whether there are timing violations, i.e., whether the slack is positive (met) or negative (violated) for the given paths under a target clock period T_{clk} [47]. The slacks of all timing paths are collected and are used



(a) Setup and hold times



(b) Gate delay and input and output slew

Figure 2.10.: Time versus voltage diagrams relevant during STA.

to calculate global timing metrics such as the *Worst Slack (WS)*, *Worst Negative Slack (Worst Negative Slack (WNS))*, and *Total Negative Slack (TNS)*. The former is the slack of the critical path. The WNS is the worst-case slack violating the timing constraints or the minimum value over all negative slacks. This is zero when all paths have positive slacks. Lastly, the TNS is the sum of all negative slacks in all timing paths.

Two checks can be conducted to ensure a synchronous design works at the target clock frequency: *setup* and *hold* [50]. These checks confirm whether the data input is stable for some time before and after the clock edge. This is to avoid incorrect data being captured or latched through the data path. The setup check determines if the synchronous input comes in time to the sequential element and is stable by a time T_{setup} before capturing the edge of the clock. The maximum setup constraints are described in Equation 2.3.

$$T_{c2q} + T_{logic} + T_{setup} \leq T_{clk} + T_{skew}, \tag{2.3}$$

where T_{c2q} is the time from the clock signal to the output of the sequential element or flip-flop *FF1*, and T_{setup} is the setup time of of the flip-flop *FF2*. Hold checks determine if the input is stable a time T_{hold} after the edge of the clock:

$$T_{c2q} + T_{comb} \geq T_{hold} + T_{skew}. \tag{2.4}$$

Setup and hold checks confirm whether the data input is stable for some time before and after the clock edge, as shown in Figure 2.10a. These checks help to avoid incorrect data being captured or latched through the path. In case of violations, different countermeasures can be approached. Setup violations can be fixed by reducing the clock frequency or the delay of the combinatorial path. In contrast, hold violations can be fixed by increasing the logic delay or decreasing the clock skew [50]. STA tools require as inputs a gate-level netlist, a technology library describing the target Complementary Metal–Oxide–Semiconductor (CMOS) technology, and initial timing constraints for the clock and primary ports. The latter are

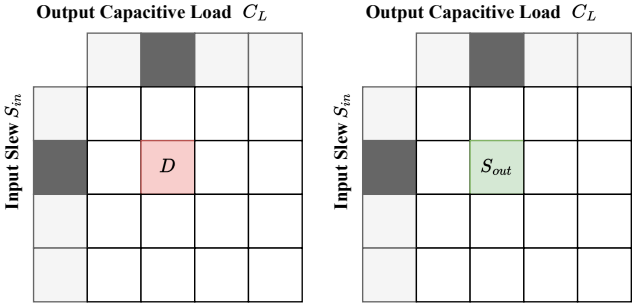


Figure 2.11.: Structure of lookup tables for gate delays D and the resulting output slew S_{out}.

commonly described using a Synopsys Design Constraints (SDC) format file, with commands and arguments standardized across different EDA tool vendors. The timing constraints in the SDC input file define the clock configuration, such as skew (uncertainty), period, and delay, as well as the delay, slew (transition time), and capacitive loads of primary ports. ATs are calculated by accumulating the delays of the gates in the combinatorial path or cloud of Figure 2.9. The *delays* through each gate are propagation delays. They are the time required for a signal to go from an input to an output pin. The pin-to-pin delays are affected by many factors, such as the functionality of the gate, fan outs, fan ins, and slews or input transition times [51]. The *fan-out* is the capacitive load at the output. The *slew* is a signal's transition time. Slews propagate along the timing path, i.e., the slew of the output pin of a gate is the input slew of the following connected components in the timing graph. As shown in Figure 2.10b, the delay is commonly measured between 50% of the input transition and 50% of the output transition. Thus, four possible timing effects exist depending on whether the input and output signals are rising or falling. The slew is measured between 10% and 90% of the signal transition time [47].

Technology library files [23] describe wire and gate delays and slews as Non-linear Delay Models (NLDMs). These models result from multiple PSPICE simulations of the gate under different operating conditions regarding the Process, Voltage, and Temperature (PVT). The technology libraries describe all available cells in a technology, their area, pins, properties, and timing models. Gate delay is modeled by characterizing the gate delay D and the output slew S_{out} as a function of the input slew S_{in} and the output capacitive load or fan-out C_L, i.e., $D = f(C_L, S_{in})$ and $S_{out} = f(C_L, S_{in})$. The technology library files use the Lookup Table (LUT) method for function approximation, as shown in Figure 2.11. Specifically, the gate delay is defined as a LUT with two variables as indexes: input slew and fan-out or capacitive load. Listing 1 shows an example of a gate definition in the technology library. During timing analysis, the STA tool uses interpolation or extrapolation to calculate delays and output slews for input values that are not specified or outside the defined indexes. The output slews are propagated and used as input slew for the following connected pin in the path [50]. Moreover, gate and wire delay models are scaled according to the current PVT conditions, which physically affect the cell behavior. The selected PVT values are considered fixed; thus, STA is deterministic.

STA is an efficient method for detecting timing violations for setup and holding checks without requiring input vectors. However, running STA tools for different RTL variants consumes time

Listing 1 Example of the description of a gate in the technology library.

```
cell(AND2) {
    pin(A) {
        direction : input;...}
    pin(B) {
        direction : input;...}
    pin(Z) {
        direction : out;
        function : "((!A)*B)";
        timing() {
            related_pin : "A";
            timing_sense : "negative_unate";
            fall_transition(lut5_5x5) {
                index_1("0.1, 0.2, 0.3, 0.4, 0.5");
                index_2("0.1, 0.2, 0.3, 0.4, 0.5");
                values( "0.5,  0.3,  0.6, 0.0,  0.7",
                        "0.3,  0.3,  0.8, 0.5,  0.7",
                        "0.2,  0.7,  0.1, 0.7,  0.9",
                        "0.5,  0.7,  0.6, 0.4,  0.9",
                        "0.5,  0.7,  0.6, 0.5,  0.9");}
            rise_transition(lut5_5x5) { ...}
            fall_delay(lut5_5x5) { ...}
            rise_delay(lut5_5x5) { ...}}
        timing() {
            related_pin : "B";
            timing_sense : "positive_unate";
            fall_transition(lut5_5x5) {...}
            rise_transition(lut5_5x5) {...}
            fall_delay(lut5_5x5) { ...}
            rise_delay(lut5_5x5) { ...}}
    }
}
```

and computation resources. STA requires obtaining a gate-level netlist for the RTL design, i.e., running (at least logic) synthesis tools. Moreover, the STA tool traverses all paths in the netlist and per each gate in each path, the tool performs inter or extrapolation of delay and slew models to get timing information of every pin-to-pin connection. Thus, a faster method is required to approximate the timing behavior of a digital design.

2.4. Summary

This chapter briefly provides background concepts relevant to this thesis, encompassing the conventional digital design flow, RTL generation, and timing analysis. First, it describes the stages of the digital design flow, from the initial specification to the chip packaging. Despite the automation of the conventional flow through EDA tools, it remains an iterative and time-consuming process, particularly prone to timing violations during sign-off, necessitating spins and corrective measures. The higher the abstraction level in the flow, there is more room for corrections and modifications. Thus, the RTL design becomes a critical stage, and yet, it is the task of the hardware designer to manually map initial specifications to RTL. This mapping is challenging, as RTL blocks can be built in many variants or microarchitectures.

Generation frameworks have been proposed to accelerate the RTL design stage. This chapter exposes the underlying methodologies and structure of MetaRTL, as this is the framework used in this thesis. MetaRTL leverages models, metamodels, and transformations to generate HDL

from informal specifications. The benefits of using MetaRTL in this work are manifold:

1. Python is the most used programming language [52] and is applied to hardware generation and ML-based EDA.
2. MetaRTL represents a design as a hierarchical tree, allowing the usage of ML techniques over design representations independently of the target HDL. This tree, called MoD, represents a design as a compendium of basic components. These components are the building blocks of any IP from CPUs to complete SoCs. Thus, learning the timing behavior of such components allows this approach to be generalizable to any target design.
3. MetaRTL provides a set of model-to-model transformations that can be exploited to explore different RTL microarchitectures.
4. Formal properties are generated, and it is possible to verify MetaRTL designs following a four-eyes principle [53]. However, formal verification is beyond the scope of this thesis.

Moreover, this chapter summarizes timing analysis concepts, particularly focusing on STA. The general focus of this thesis is the acceleration of the conventional digital design flow following a shift-left strategy, i.e., emphasizing early analysis and feedback. Specifically, this thesis proposes a timing estimation engine at the RTL design stage. The question of how to use RTL generation frameworks, their IRs and transformations, and ML models to predict timing metrics and find optimal RTL microarchitectures has never been addressed before this thesis.

3. Machine Learning Methods

ML is a subfield of AI dedicated to developing computer algorithms that learn from data. Inspired by biological brains, ML enables machines to learn from previous experiences to make informed predictions or intelligent decisions without requiring hard-coded rules [54]. Even though the term machine learning was introduced in 1959 by Arthur Samuel [55], it keeps revolutionizing industries and societies every day in more surprising manners, from fields such as drug discovery [56], generative models [57], autonomous driving [58] and robotics [59].

A standard classification of ML techniques in the literature focuses on how the algorithms leverage data during training. There are three main categories. In *supervised learning*, the data x and the expected outputs or ground-truth labels y are available. The models aim to learn the mapping of known x and y to generalize to unseen data. In contrast, during *unsupervised learning*, algorithms are provided only with input data x, and their goal is to find the underlying distribution or main patterns. Finally, *semisupervised learning* algorithms only have a portion of the target outputs y available. The latter category is common in real-life applications, where labeling is expensive or time-consuming [54].

Thanks to the availability of data provided by EDA tools during the digital design flow, this thesis employs supervised learning to solve regression tasks. This thesis proposes using ML models to approximate the timing design metrics of RTL designs. An ML-based flow aims to minimize turnaround times, resource consumption, and license utilization at the cost of less accuracy in the estimations [60]. Beyond that, this thesis provides mechanisms to annotate estimated delay at RTL, one level above gate netlists, the usual place for delay annotation. This chapter provides a gentle introduction to ML, focusing on the flow needed to solve a regression task. To this end, the building blocks of such flow are described in detail, mentioning methods and tools employed in this thesis.

3.1. Machine Learning Flow to Solve Regression Tasks

Regression models are trained in supervised learning to learn the relationship between data x and ground-truth labels y. Generally, x is a vector of multiple discriminative features describing the data, and y can be a single or multiple continuous value. The input to a regression model is a feature vector $\mathbf{x}_n$ where $n \in \mathcal{N}$ represents the total number of features. The model's output or ground-truth label $\mathbf{y}_m$ is a vector representing the $m \in \mathcal{M}$ regression objectives. During training, a dataset with a total number of D samples is used, each described by feature and objective vectors x and y. The trained model aims to predict the output vector y' for unseen samples x'.

Solving a regression task with ML models requires several steps, as outlined in Figure 3.1.

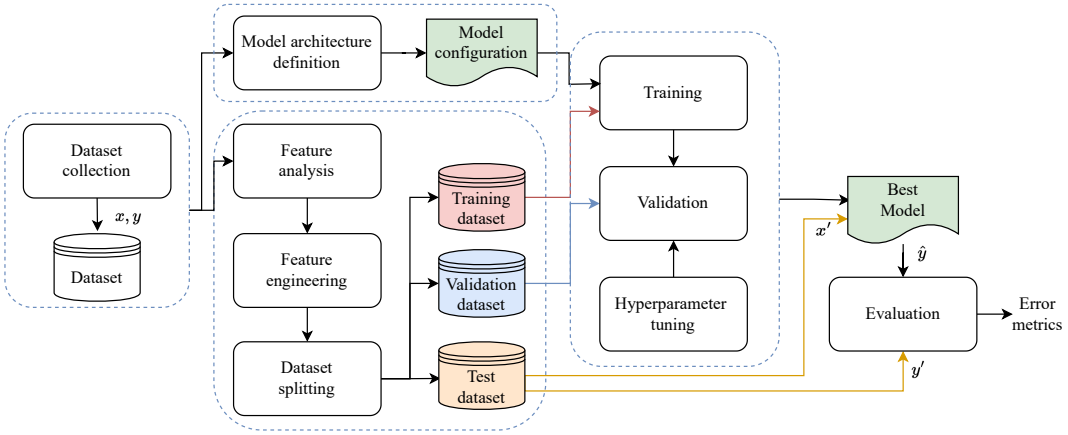


Figure 3.1.: Building blocks of the ML flow for supervised learning.

First, data samples representing the problem's nature and covering most corner cases must be collected. Features are analyzed and selected from the initial data based on their ability to represent different patterns and discriminative data characteristics. These features can be categorical or numerical representations with varying degrees of precision. However, since ML models require a numerical input, engineering, and preprocessing features are necessary steps. Next, datasets containing features and objectives are built and split for training, validation, and testing. ML model architectures that suit the training data structure are then selected, and the models' hyperparameters are tuned and trained. Finally, trained models are evaluated over unseen data in the test dataset. The ML flow outputs the best model capable of predicting the output y' of any unseen data x', along with error metrics that measure how well the regression model performs for the selected test set. More details on each of these steps for a regression task are provided in the following subsections.

3.1.1. Dataset Collection

Supervised learning methods are data-driven. Thus, the quality of the predictions depends on the quality of the training data. Data collection involves gathering information from different sources and building structured datasets. This process also involves analysis of which data best represent the nature of the task and data cleaning to remove noise that may mislead the model. Alternatively, data augmentation is used if available data are limited. For EDA problems, data are available, and noise is neglected as tool results are taken directly as ground-truth labels. However, extra effort is required to ensemble a set of circuits with its respective ground-truth labels. Data collection is a critical step in developing a successful supervised learning model.

3.1.2. Feature Analysis

Meaningful datasets of features should be constructed from the raw collected data. This step is crucial for ensuring that the resulting dataset accurately represents the underlying patterns and relationships in the data. To that end, statistical approaches analyze the correlation and

importance of data features to guide the selection. Some of the approaches described here are used in this thesis:

3.1.2.1. Spearman's Rank Correlation Coefficient

One method to analyze discrete and continuous variables is the correlation between features. A particular case of correlation is called Spearman's Rank Correlation Coefficient (SRCC) [61]. It measures how well a monotonic function can represent the relationship between two variables X and Y, which may be linear or not. An SRCC of one indicates that two variables increase in value together. An SRCC of -1 corresponds to a decreasing monotonic trend: as one variable increases, the other decreases. The formula for the SRCC, denoted as ρ, can be expressed as:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}, \tag{3.1}$$

where d_i is the difference of the ranks between two pairs of samples, i.e., $d_i = R[X_i] - R[Y_i]$, and n is the number of samples in the dataset.

3.1.2.2. Mutual Information

The Mutual Information (MI) [62] between two discrete variables, X and Y, measures their linear and non-linear dependencies. It measures how much knowing one of these variables reduces the uncertainty about the other. Specifically in ML and feature analysis, the MI helps in understanding the dependency between a vector of features X and the vector of target variables Y. A MI of zero indicates that the features and target variables are independent. In other cases, the MI is positive, and the higher the value is, the higher the dependency. The formula for the MI in terms of the probabilities of X and Y is:

$$I(X, Y) = \sum_{y \in Y} \sum_{x \in X} P(x, y) \log \frac{P(x, y)}{P(x)P(y)}, \tag{3.2}$$

where x and y correspond to single elements of the feature and target vectors, respectively. $P(x, y)$ is the joint probability distribution of X and Y, i.e., the probability that $X = x$ and $Y = y$ at the same time. Lastly, $P(x)$ and $P(y)$ are the marginal probability distributions of X and Y, respectively. The latter are the probabilities of observing x and y independently. MI is also represented as the entropy of X minus the entropy of X given Y. In [62], these entropies are estimated using K-Nearest Neighbors (KNNs) [63] distances over the permutation of the features X. The KNN algorithm is a non-parametric method that searches the k-closest instances to an unseen data point through the training set and returns the closest data as the prediction.

3.1.2.3. Shapley Values

Shapley values [64] are a game theory concept used for explainable and interpretable AI [65]. Specifically, Shapley regression values are used to measure the importance of features in linear

models. This is achieved by training two models: one with the chosen feature and one without it. The final Shapley values are determined by comparing the performance of both models and computing the differences for all possible feature subsets. Precisely, the Shapley values are calculated by taking a weighted average over all possible subsets [65].

3.1.3. Feature Engineering and Preprocessing

Once the most relevant features from the data are selected, further steps are performed to convert raw features into a more suitable representation that can be passed through ML functions. Feature engineering describes actions that can be performed on different feature types to cope with a given ML model. These actions include the transformation and generation of domain-specific features [66].

Features can be numerical, covering integers, floats, binary values, or categorical. Numerical features can also be pseudo-continuous or discrete. ML techniques for regression usually expect features and objectives to be numerical and standardized, i.e., in a similar scale of magnitude. Thus, different preprocessing methods are applied, from transformers and scaling to normalization methods. Scalers transform the data to a range determined by a maximum and minimum level. Normalization aims to transform the data to have a unit norm. Transformers apply linear or non-linear functions to the features, such as discretizing continuous features into bins, quantiles, or power transforms [67].

The input features for the first layer should be numerical or one-hot encoded. Categorical features are often numerically represented using ordinary and one-hot encoding formats, as shown in Figure 3.2. The former uses one ordinal number per category. The latter, also known as *one-of-K* or dummy encoding, employs binary vectors of size equal to the total number of categories. In one-hot encoding, each categorical feature is represented with a sparse vector of zero elements containing a single one in the position corresponding to the given category [67].

Another alternative is to use embedding vectors [68]. Embedding vectors are widely used in Natural Language Processing (NLP) to embed words and to learn similarities between them [69], [70]. From ordinal numbers, Neural Networks (NNs) are trained to generate latent or embedding vectors per each category. The generated vectors are not sparse but instead represent which data and categories are closer in a latent space. The size of the embedding vectors can be higher or lower than the number of categories. Thus, feature embeddings alleviate two key issues of one-hot-encoded vectors: high dimensionality and sparsity.

3.1.4. Model Architecture Selection

The next step in the ML flow is the model selection. This thesis employs vector- and graph-based models to cope with the data structure of different generated datasets. Tabular datasets are processed with shallow or deep models, like MLP networks. Graph-based datasets are preprocessed using GNNs [71]. Details about these three types of architecture are provided below.

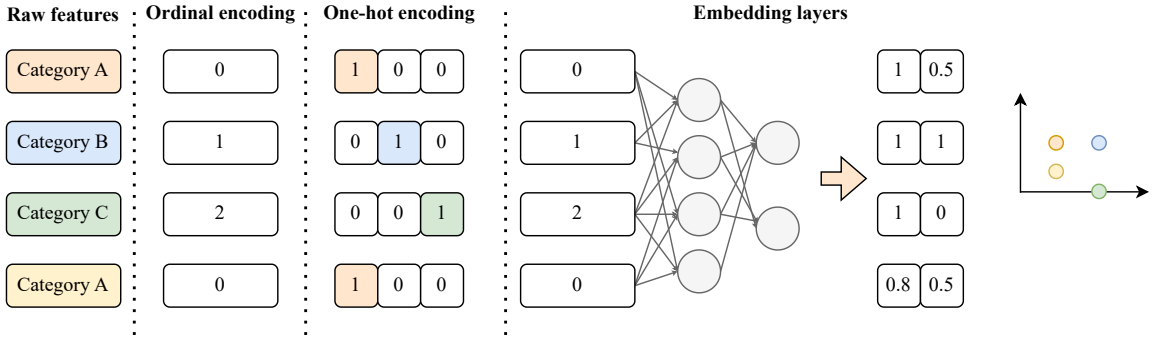


Figure 3.2.: Overview of different encoding formats for categorical variables.

3.1.4.1. Shallow Models

These classical models are suitable for tasks with simple patterns and, thus, do not require complex representations or pattern abstractions. Shallow models require hand-crafted features for performing regression or classification. Examples of shallow models are decision trees and random forests [72].

Decision trees are commonly used for their simplicity in being visualized, interpreted, and implemented. They are used for classification, decision-making, and regression tasks when the target output is continuous. Decision trees can be seen in programming languages as nested *if-else if-else* instructions. Every node in a tree represents a condition on a single feature. The arcs coming from a node are labeled with the possible values or options for that feature and lead to a subordinate decision node with a different feature or to the target output y. Figure 3.3 shows an example of a decision tree on the top, where the nodes and arcs are highlighted in green to represent the conditions the input data meets, leading to an output value highlighted in red.

One way to improve the performance of decision trees is to use ensemble methods such as random forests or boosting. Random forests ensemble multiple decision trees at training time to create a more robust and accurate model. Figure 3.3 shows a random forest with three trees at the bottom, each with different nodes and output values. Similarly, the green nodes and arcs represent the conditions the input data meets along each tree. For regression tasks, the output of the random forest is the mean of the outputs of each tree.

3.1.4.2. Multi-layer Perceptron Networks

Artificial NNs are computational models inspired by neuroscience. The main feature of NNs is their ability to process information and learn by example. Therefore, they are used for a wide variety of problems that cannot be described by any task-specific rules but with available training data. The most simple example of a Deep Learning (DL) model is the MLP [73] network, which can represent various functions using appropriate parameters. Deep models are called *deep*, as they consist of multiple hidden layers, allowing automatic abstraction and

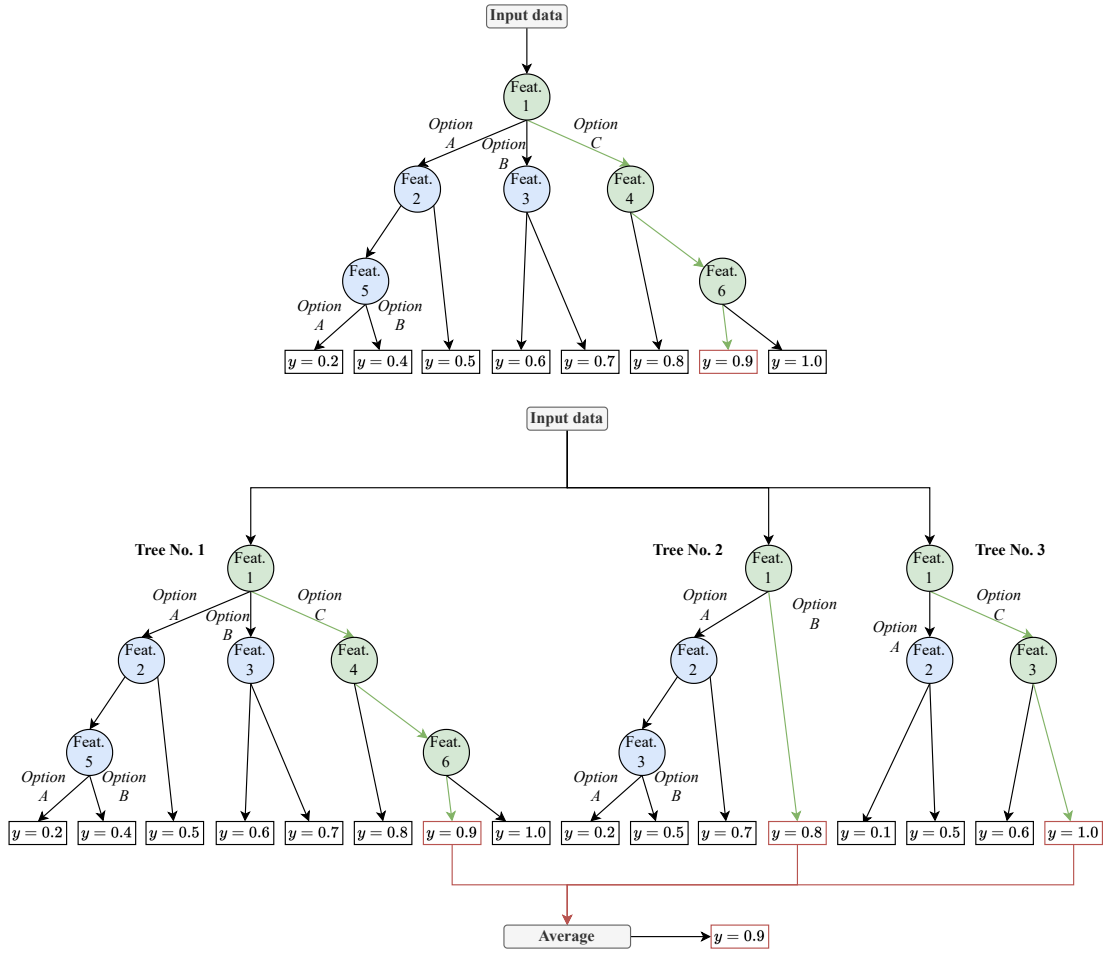


Figure 3.3.: Decision tree (top) and an ensemble of trees to compose a random forest (bottom).

learning from underlying data patterns. Thus, they are suitable for more complex tasks such as classification or regression. Finally, they scale better to large datasets.

Inspired by biological neurons, an artificial neuron receives inputs and produces output signals, as shown in Figure 3.4. The input signals $[x_0, ..., x_n]$ are scaled by learnable factors called weights $[w_0, ..., w_n]$, representing the strength of one neuron's influence on the other. All the scaled inputs are added up $\sum_i^n w_i x_i$ inside the neuron. In biological neurons, if this sum is greater than a threshold, the neuron will fire. In the computational model, the firing is controlled by a bias term included in the sum. This sum is passed through a non-linear activation function $\sigma(\cdot)$, which produces the output signal [74].

Figure 3.4 also shows an NN as a collection of interconnected neurons often distributed into different layers. An MLP consists of three layer types: input, hidden, and output. The input layer consists of as many neurons as the number of input features. Generally, the first layers are responsible for the feature and pattern extraction in the input data. The last layers take those features and learn how to classify or approximate them. The number of hidden layers and neurons are hyperparameters to be tuned. The output layer has neurons equal to the number

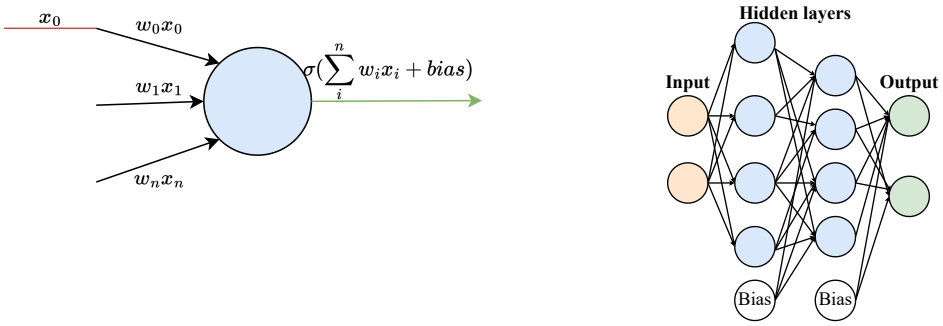


Figure 3.4.: Mathematical model of a neuron and layers of an NN.

of objectives or classes to predict in a regression or a classification task, respectively. The layers are fully connected, meaning each element in the subsequent layer results from applying the activation function to the weighted sum of all values in the current layer. NNs use the backpropagation learning method to update all weights during training.

3.1.4.3. Graph Neural Networks

The traditional approach for processing graph-structured data is to use shallow embedding methods. These aim to decompose the node information into low-dimensional embedding vectors that consider the position of the nodes in the graph and the structure of the neighborhood [75]. One of the best-known graph embedding techniques is Random Walk [76]. Given a starting point within a graph, this technique selects a random neighbor point. As a second step, a neighbor of the randomly selected point is chosen again. This is continued recursively and generates a random sequence of points, namely the random walk. DeepWalk [77] and Node2vec [78] are well-known graph embedding methods based on random walks. Although these methods have achieved groundbreaking success, they have some limitations [75]:

- Their encoders, which map essential information from the graph to an embedding space, optimize a unique embedding vector per node. This can be computationally and statistically expensive in large graphs.

- They are *transductive*, i.e., they can generate embeddings only for nodes seen during training. This is a significant drawback, as the model cannot generalize to unseen nodes.

- They do not consider node features that could provide precious information during the encoding.

Similarly, for two main reasons, other NNs, such as Convolution Neural Networks (CNNs), cannot operate directly on graph data without mapping their structure to a fixed-size vector format. First, graphs do not have a fixed notion of locality or sliding window to perform the convolution operation. Second, a graph does not have a fixed order of nodes. Operations acting over graphs should be invariant to the node ordering or permutation. To overcome the

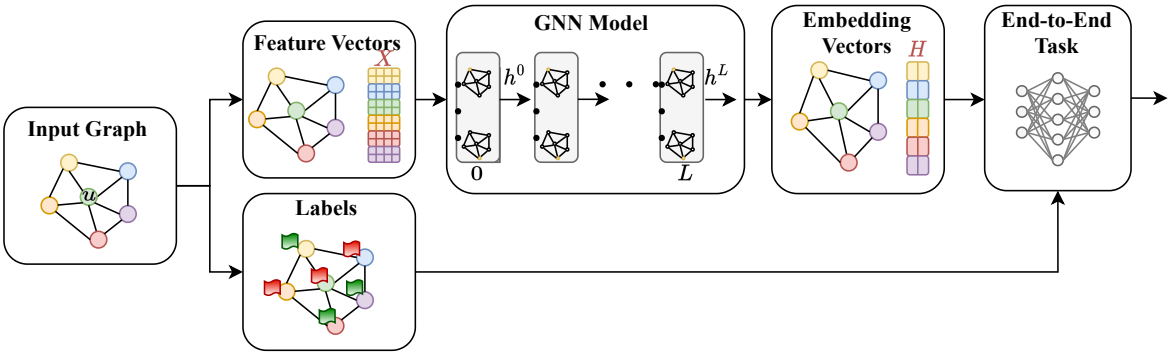


Figure 3.5.: General pipeline when using a GNN.

limitations of the shallow methods and deep NNs, GNNs are introduced by Gori et al. [71].

GNNs are deep NNs that process graphs as inputs and learn to represent them as embedding vectors $\mathbf{h_u}$ for all nodes in the graph. These embedding vectors $\mathbf{h}$ capture feature information of a node and its neighbors and, thus, implicitly topological information. GNNs employ a *message-passing* strategy in which the node embeddings or messages are passed through a neighborhood. The number of GNN layers L used determines the distance for which the messages are propagated. The propagated messages are transformed using a non-linear differentiable function $\sigma(\cdot)$ to update the node embedding vectors. The primary update formula of a GNN is given in Equation 3.3:

$$h_v^{l+1} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^l}{|N(v)|} + \mathbf{B}_l h_v^l), \tag{3.3}$$

where $\mathbf{W}$ and $\mathbf{B}$ are trainable weight matrices. By stacking all embedding vectors $\mathbf{h}_u$ for all nodes in the last layer h_v^L, the matrix of embedding features $\mathbf{H}$ is obtained, where $\mathbf{H} \in \mathbb{R}^{N \times d}$ and d is the desired size of the embedding vector. The feature vectors are the initial node embeddings in the first layer, $h_v^0 = x_v$, and thus $\mathbf{H}^0 = \mathbf{X}$.

The general overview of a GNN pipeline is shown in Figure 3.5. The matrix $\mathbf{H}$ is the input to an MLP model, which learns to perform regression or classification at the edge, node, or graph level. The flow of this pipeline is influenced by three main factors: the graph definition, the selected graph type, and the GNN architecture chosen to handle the input graphs and the end-to-end task. These factors can significantly affect the performance of the GNN:

3.1.4.3.1. Graph Definition A graph is a data structure representing interactions between related objects. Mathematically, it is as a tuple $G = (V, E)$, where $\mathcal{V}$ is the set of nodes, $N = |\mathcal{V}|$ is the total number of nodes, and $\mathcal{E}$ is the set of edges. The edges are defined as the connection between nodes, e.g., for the nodes $u, v \in V$, the edge is represented as $e_{u,v} \in \mathcal{E}$. The neighborhood of a node v is defined as $\mathcal{N}(v) = \{u \in V | (u, v) \in \mathcal{E}\}$. A graph can be represented then as a list of nodes and edges. However, a more convenient representation is through an *adjacency matrix* $\mathbf{A}$ defined as $\mathbf{A}_{u,v} = 1$ if $e_{u,v} \in E$, otherwise $\mathbf{A}_{u,v} = 0$. The degree of a

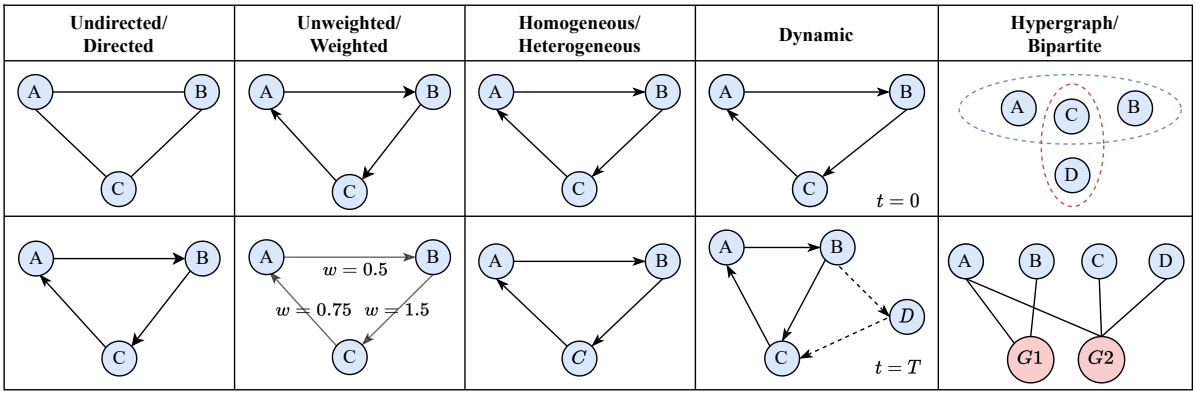


Figure 3.6.: Overview of different topologies of graphs.

node is the number of nodes D incident to a node u. Mathematically, it can be defined as $D_u = \sum_{v \in V} \mathbf{A}_{u,v}$.

For exploiting GNNs the most, ***featured graphs*** are employed. Mathematically, a featured graph is an extended tuple $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \mathbf{X}, \mathbf{X_e}\}$, where the matrices $\mathbf{X}$ and $\mathbf{X_e}$ represent the features for all nodes and edges, respectively. The matrix $\mathbf{X}$ is built by stacking all vectors of features per node $\mathbf{x_u}$. Thus, $\mathbf{X} \in \mathbb{R}^{N \times D_n}$. Similarly, the matrix $\mathbf{X_e}$ represents the matrix of edge features and $\mathbf{X_e} \in \mathbb{R}^{N \times D_n}$. D_n and D_n signify the dimensions of the feature vectors per node and edge [75].

3.1.4.3.2. Types of Graphs Figure 3.6 illustrates diverse types of graphs, where the colors of the nodes represent their type, i.e., the red and blue nodes are of different types. All depicted graphs are *static*, except for the *dynamic* graph in the fourth column, where the structure changes with time. Lastly, G_1 and G_2 represent *hyperedges*. If the direction of the edges matters, the graph is *directed*, and the adjacency matrix $\mathbf{A}$ is symmetric. If the edges do not have a direction, the graph is *undirected*. In case the edges are represented with a cost or real value, the graph is called *weighted*, and $\mathbf{A}$ will have real values as entries. *Multiplex* graphs can be decomposed into layers, and the relation between each layer and the belonging nodes are additional *intra-layer* edges. Graphs can also be *homogeneous* or *heterogeneous*. In the former, all nodes and edges have the same type. In the latter, nodes have different types, and their attributes may be of distinct types [75]. In *dynamic* graphs, the feature and topological information vary with time. In other cases, they are called *static* graphs [79]. These categories of graphs are orthogonal, i.e., they can be combined.

GNN architectures support several categories of input graphs, such as heterogeneous [80], dynamic [81], directed [82], hypergraphs [83], and bipartite graphs [84]. Research in the field aims to extend GNNs to cope with different topologies and dimensions of node, edge, and graph features.

3.1.4.3.3. Classification of GNNs Novel architectures have inherited the message-passing strategy and are commonly divided into four types [85]. First, Recurrent Graph Neural Networks (RecGNNs) process the node information recurrently by assuming that the nodes exchange information with their neighbors until a stable point is reached. Second, Convolutional Graph Neural Networks (ConvGNNs) aim to learn embedding vectors by stacking multiple graph convolutional layers. Unlike RecGNNs, where weights are shared, ConvGNNs use different weights per layer. ConvGNNs generalize CNNs to graph data and are the most commonly used GNNs. Moreover, Graph Autoencoders (GAEs) belong to the family of unsupervised frameworks, as training data have no ground-truth labels. Thus, the computed loss depends on the topological information of the entire graph, including node and edge features [86]. GAEs are employed in graph-based representation learning and graph generation. In both tasks, an encoder calculates the corresponding embedding vectors h_u for every node $u \in V$ using ConvGNN or RecGNN layers. These graph embeddings extracted by the encoder are low-dimensional vectors that hold node features yet retain the graph's topology information [85]. The embeddings serve as inputs to the decoder, reconstructing the adjacency matrix $\hat{\mathbf{A}}$ so that the distance to the original matrix $\mathbf{A}$ is minimized. Finally, Spatial-Temporal Graph Neural Networks (STGNNs) address a wide range of problems where the structure of the input graph changes over time, i.e., the matrices $\mathbf{A}$ and $\mathbf{X}$ change along the time axis.

3.1.5. Hyperparameter Tuning

The configuration of an ML model implies the definition of a set of parameters that cannot be learned during training, such as the number of neurons, layers, or learning rate. These are called hyperparameters. Using proper values for those hyperparameters affects the model's performance and the learning process's efficiency. Thus, many techniques have been studied to find optimal values and fine-tune ML models easily. In general, hyperparameter optimization is represented in equation form as $\hat{x} = argmin(y)$, where $y = f(x)$ represents the objective function to minimize, $\hat{x}$ is the set of hyperparameters that yields the lowest value of the objective function, and x takes any value in the space defined in X.

Methods to optimize objective functions have been well-studied. Manual tuning, grid search, and random search are considered brute-force methods. Grid search is an exhaustive approach to evaluate the hyperparameter values defined in a grid. The random search approach will randomly choose the values for the hyperparameters. Grid and random search perform better than manual tuning but do not consider past evaluations. Hence, they lose significant time evaluating hyperparameters in the same inadequate region. Furthermore, the objective functions to minimize are usually computationally expensive, and running them each time with a new set of parameters is time-consuming [87]. To overcome this, Bergstra et al. [87] introduce a model-based method called sequential model-based optimization. The main idea of this approach is to create a surrogate model of the objective function using a probabilistic distribution. This surrogate model is based on Bayes' theorem in Equation 3.4.

$$p(y/x) = \frac{p(x/y)p(y)}{p(x)}. \tag{3.4}$$

There are various methods to construct a surrogate. One such approach is presented by

Bergstra et al. [87] as a Tree-Structured Parzen Estimator (TPE). This particular approach models the probabilities $p(y)$ and $p(x/y)$ instead of $p(y/x)$. It determines the probability of the hyperparameters x given the objective function y value. Equation 3.5 shows the probability $p(x/y)$, which can be either $l(x)$ or $g(x)$ depending on whether y is less than or greater than a threshold $\hat{y}$.

$$p(x/y) = \begin{cases} l(x), & \text{if } y < \hat{y} \\ g(x), & \text{otherwise.} \end{cases} \tag{3.5}$$

The TPE approach builds the surrogate by taking samples from $l(x)$ and choosing the best hyperparameters using the ratio between $l(x)$ and $g(x)$. This surrogate is a computationally cheaper version of the objective function. Consequently, it is employed to identify the best-performing hyperparameters, and only the best configurations are evaluated in the expensive objective function. The resulting evaluations are then used to update the surrogate model. In this way, Bayesian methods consider past evaluations and choose new hyperparameters more smartly without evaluating the objective function each time. This approach significantly reduces the computational cost and time for hyperparameter optimization.

3.1.6. Training, Validation, and Inference

After the preprocessing and transforming, the data are split into training, validation, and test datasets. These are used in the respective distinct phases in the lifetime of MLP or GNN models [88]:

The **training** phase is the learning process, consisting of a forward and a backward pass. In the former, the model takes a labeled input from the training dataset and calculates the activation values from one layer to the next on the right, using randomly initialized weights. In other words, given a specific input and for a fully connected layer, the output values of each neuron are calculated, as shown in Figure 3.4. At the output layer, the model calculates the *loss* or error between the actual output and the ground-truth label of the input. This error tells how well or poorly the model performed for the given input.

In the backward pass, the model propagates this error, calculating the gradient of the loss function with respect to the weights. This is done for each layer, iterating from the last to the first hidden layer. Using an *optimizer* algorithm, the weights are updated to minimize the loss and to reduce the difference between the actual and the intended output. The update is proportional to a *learning rate* or step factor to control the step size. The forward and backward passes are repeated for N samples of a *training dataset* and E epochs. In this way, the model uses the loss to adjust the weights for each layer, producing an output closer to the ground-truth value. The loss function, the optimizer strategy, and the learning rate are commonly hyperparameters.

Validation is performed after each training epoch to evaluate how well the training progresses when the model is run over the unseen validation dataset. The validation error is calculated but not used to update the weights in the backward pass. By comparing the training and validation loss curves, it is possible to detect whether the model is *underfitting* or *overfitting*. Underfitting

occurs when the model is too simple to capture the underlying patterns in the training data. This results in high training and validation errors. On the other hand, overfitting occurs when the model is too complex and learns the training distribution very well but fails to generalize to unseen data in the validation set. Ideally, the model should be complex enough to capture the underlying patterns in the training data but not overfitted to them. Different techniques to address under and overfitting are available. For instance, *dropout* is used during training to overcome overfitting by randomly dropping out or deactivating a set of neurons. The ratio of the neurons to be dropped is called the *dropout rate*.

The validation dataset is also used to fine-tune hyperparameters, as using the training data for this purpose would bias the search. The best combination of hyperparameters can be selected by changing hyperparameters and evaluating the model's performance on the validation dataset. This helps to ensure that the model generalizes well to new, unseen data.

The **inference** phase is similar to the forward pass described above but using the test dataset, i.e., new and unlabeled input data. This implies that no loss can be calculated and no backward pass is needed. The model is solely used in this phase, and it can be verified if it properly learned the task during the training phase.

During training, validation, and testing, the loss and performance of the model are assessed using standard *error metrics* [89]. The most commonly used loss in a regression task is the Mean Squared Error (MSE). This averages the squared value of the *residuals* or the difference between the actual and predicted values, as shown in Equation 3.6.

$$\mathrm{MSE}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{3.6}$$

where y_i is the ground-truth value and $\hat{y}_i$ the predicted one. This metric squares the residuals to get the model's performance despite negative values and penalizes higher errors. Thus, this metric is also sensitive to the outliers. The Mean Absolute Error (MAE) in Equation 3.7 copes better with data outliers and is the average absolute difference between actual and predicted values.

$$\mathrm{MAE}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|, \tag{3.7}$$

The lower the values of MSE and MAE, the higher the prediction accuracy of the regression model. Other error metrics are used to measure the performance of a regression model, such as the coefficient of determination R^2, the Mean Absolute Percentage Error (MAPE), the Root Mean Square Relative Error (RMSRE), and the Maximum Error (ME) described in Equation 3.8, Equation 3.9, and Equation 3.10, respectively. The R^2 in Equation 3.8 measures how well the data fits the model compared to the mean value $\bar{y}$. The higher the R^2 value, the better the model is. The R^2 can also be negative, indicating that the mean of the data is a better estimation than the prediction. The MAPE is not sensitive to scaling, as it calculates relative error with respect to the actual output. An arbitrary small ϵ is defined to guarantee defined results when $y_i = 0$. The ME computes the maximum residual error and provides

information about the worst difference between the actual and predicted values [89].

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y_i})^2}, \tag{3.8}$$

$$\text{MAPE}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}\frac{|y_i - \hat{y}_i|}{\max(\epsilon, |y_i|)}, \tag{3.9}$$

$$\text{ME}(y, \hat{y}) = \max(|y_i - \hat{y}_i|) \tag{3.10}$$

3.2. Summary and Tools

Figure 3.1 summarizes the steps when solving a regression task using supervised ML techniques. In general, collected datasets are passed through a feature analysis and engineering to select and build relevant and meaningful patterns of the data. Moreover, features must be transformed into numerical discrete or continuous variables in standard or fixed ranges. After preprocessing, data is split into training, validation, and testing. The data structure and complexity determine the ML model architecture to be chosen. Thus, shallow and deep models are built to process Euclidean or graph-based data. Hyperparameters are fine-tuned using search algorithms, such as random or TPE search. Afterward, training, and validation data are employed during the training process. Pre-trained models are used for inference over unseen data in the test dataset. Finally, regression error metrics calculate the model's loss and performance.

This thesis maps the estimation of timing metrics of digital designs to a regression task. This is solved following the steps in Figure 3.1 and using the methods, steps, and metrics described in this chapter. Initially, data collected from EDA tools are built as tabular datasets, and shallow and MLP models are trained. However, this does not consider that many EDA objects are represented naturally as graphs. Thus, the modern GNNs are explored as an opportunity to solve EDA problems directly using graph structures for circuits, RTL IRs, and netlists.

The widespread use of ML models across different fields can be associated with available free and open-source ML frameworks and tools. This thesis leverages Python as a programming language and Pytorch [90] as a robust ML framework. Pytorch defines models as dynamic graphs, allowing the programmer to create, modify, and execute nodes on the fly. PyTorch facilitates the backward pass thanks to its automatic gradient computation system called *autograd*. This automatic differentiation only records the parts of the graph that are relevant for the output computation rather than the entire graph. It also allows in-place operations and uses core-logic modules written in C++. Moreover, it provides an array-based programming model with GPU acceleration. These features boost and optimize the dynamic, eager execution of graphs. PyTorch is widely used in research and academia as it speeds up the model design and eases debugging.

The Pandas [91] package is employed to handle tabular data structures. Pandas eases the cleaning and splitting of datasets and the preprocessing stages in conjunction with the Scikit-learn package [89]. This thesis also employes the latter to build shallow ML regressors and evaluate

different models using the available implementation of standard error metrics. When collecting graph-based datasets, the Graph Modeling Language (GML) [92] format is employed to map the RTL IRs to graphs in a file. Moreover, the NetworkX package [93] is employed to read, write GML files and process them. NetworkX provides useful functions to find predecessors, successors, and paths between two nodes. For building GNN-based models and manipulating graphs, the Deep Graph Library (DGL) [94] package is used. DGL is framework-agnostic, i.e., it can be used on top of TensorFlow or PyTorch. Finally, this thesis uses the HyperOpt library [95] to instantiate a hyperparameter optimizer using random, grid search, or TPE.

4. Related Work

The great success of ML in solving NP-complex problems, classification, and regression tasks in other fields has drawn the attention of the EDA community [96]. Thus, ML has been used across the different stages of the design flow for analog and digital circuits targeting different hardware platforms and processes. Specially, Huang et al. [96] recognize four main areas in how ML can be used: predicting optimal configurations, approximating design metrics without running EDA tools, optimizing PPA by exploring the design space, and optimizing designs through Reinforcement Learning (RL) as in the envisioned future with an AI-assisted design flow. Moreover, they highlight examples of ML applications in each category and the digital design flow's different stages, platforms, and tasks.

This chapter is split into three sections. First, a set of ML applications to the digital design flow are briefly listed to exemplify the gap of applications using designs at the RTL as inputs. This gap motivates this thesis and recently published work in the field. Second, related work estimating timing-related metrics are summarized and categorized based on the methods employed, i.e., non-ML and ML-based. Like this thesis, ML models are explored based on the type of data structures used for training, i.e., Euclidean or graph-based structures. Finally, this chapter is summarized, and selected work from the literature are directly compared to the methods of this thesis.

4.1. Gap of Machine Learning Applications to the RTL Design Stage

Table 4.1 summarizes related work applying ML to the digital design flow. Most of the approaches extract features from gate-level netlists after logic synthesis or placement and aim to predict objectives of a more costly stage, such as routing. The higher the level of abstraction, the more challenging it is to use ML models. Thus, there is a gap in ML applications at the early design stages, as reflected in Table 4.1, between logic synthesis and HLS. This section summarizes ML applications at the physical design stage and earlier.

4.1.1. Physical Synthesis

ML applications have been used to predict design metrics or optimize the synthesis steps of floor planning [97], placement [98]–[100], and routing [101], [112].

Table 4.1.: Summary of related work applying ML to the digital design flow.

Physical Synthesis	Logic Synthesis	RTL Design	High-level Synthesis	Abstract Specifications
[97]–[101]	[102]–[104]	[105]–[107]	[108], [109]	[110], [111]

In floor planning, Google presents an RL framework for the macro placement of Tensor Processing Units (TPUs) [97]. They incorporate a GNN into the RL framework to encode the different states of the process. This framework predicts the reward labels for congestion, density, and wire length while also being able to generalize to unseen netlists. As a result, the RL agent gives equal or better results than a human designer but takes hours instead of months. The latter results have raised much discussion in the community [113], [114]. In particular, it has been demonstrated that classical methods, such as simulated annealing, outperformed Google's RL method. As a side effect, research in the field is becoming more robust and transparent.

During placement, Xie et al. [98] estimate the net length obtained as the half-perimeter wire length of the net's bounding box after placement. Moreover, Lu et al. [100] target optimization by guiding placer tools to make beneficial decisions. Using GNNs, they generate clusters of cells based on the hierarchy information and their affinity with memory blocks. The resulting clusters are the optimal placement groups of a netlist and are used as guidance for any placer tool. Lastly, Agnesina et al. [99] formulate a general framework mapping any PPA optimization task in EDA to an RL problem, like the macro placement optimization presented by Google. As a proof of concept, this framework is applied to the wire length optimization during 2-D placement.

Using RL, two other frameworks are proposed to optimize placement (DeepPlace) and placement and routing together (DeepPR) [112]. DeepPlace aims to learn the optimal placement of macros and standard cells, while DeepPR learns optimal macro placement and routing. Similarly, using post-placement gate-level netlists as inputs, Kirby et al. [101] use GNNs to predict their congestion values in the routing stage. ML techniques have also been applied to subsequent stages of the physical design, such as during functional reverse engineering [115] and testing [116], to insert fewer test points on the design while maximizing the fault coverage.

4.1.2. Logic Synthesis

Inspired by the many possible optimizations during logic synthesis, ML has been used to automate the search for the optimal logic optimization recipe. Yu et al. [102] train an ML model to learn synthesis flow features that generate optimal QoRs for a target IP. They use ABC [117] and its available logic transformations. Possible combinations of transformations or synthesis recipes represent the inputs to a CNN, which predicts the QoR levels for a given combination. Similarly, LSOracle [103] uses ML models to select which optimizer should handle different design parts as an And-Inverter Graph (AIG) or as a Majority-Inverter Graph (MIG). Lastly, DRiLLS [104] uses RL to find the optimal sequence of synthesis commands while targeting less area and given timing constraints.

4.1.3. High-Level Synthesis

From a higher level of abstraction, Ustun et al. [108] train a GNN to predict the mapping and clustering of HLS arithmetic operators in FPGAs. Primarily, they map Digital Signal Processors (DSPs) and carry blocks to predict a more accurate delay for FPGA blocks. Similarly,

Pyramid [109] collects a dataset of HLS designs and synthesis reports for predicting optimal timing and resource utilization.

4.1.4. Abstract Specifications

Abstract specifications describe the requirements of an IP using natural language and serve as inputs to RTL generation frameworks. Servadei et al. [110] use ML to estimate and optimize the design cost of specific hardware IPs when mapped to FPGA platforms. They use the Hardware Software Interface (HSI) component as the case study and define the hardware and software cost in terms of area, using metrics such as the number of LUTs, Configurable Logic Blocks (CLBs), as well as flip-flops, and firmware performance, thereby considering the number of software cycles and size. The design cost represents the reward or objective function to minimize using RL or a Genetic Algorithm (GA). Similarly, Zennaro et al. [111] propose ML models to estimate the area of the control and status register interfaces for FPGAs.

4.1.5. RTL Design

At the RTL, ML has been used almost exclusively to estimate other design metrics such as power, timing, and area. For instance, PRIMAL [105] predicts power consumption for ASIC designs. Its primary disadvantage is that it is design-specific, i.e., a new model must be trained to estimate the power consumption of a new design. To overcome this, GRANNITE [106] targets RTL power estimation using GNNs and gate-level netlists as inputs. MasterRTL [107] has recently been presented as a pre-synthesis PPA estimation framework. They map the HDL code to a bit-level representation and estimate TNS, worst-case slack, and power consumption. Other applications estimating timing-related metrics arise during the development of this thesis. These contributions are studied in Section 4.2.2 as they directly relate to this thesis.

4.2. Methods to Predict Timing Metrics

State-of-the-art research has recognized the need for a faster method providing timing metrics of digital designs rather than running synthesis or STA tools. Kahng et al. [60] recently outlined how ML predictive models are increasingly used for design closure and sign-off. Some work focus on component delay estimation, as this is the minimal information required to be propagated along timing paths. Others directly estimate a design's maximum ATs or slacks given a target technology. Moreover, some studies take the RTL design as input, while others use a more concrete gate-level netlist obtained after placement or routing. Some work target FPGA platforms instead of ASIC designs. This section focuses on their methods for estimating timing-related metrics. Accordingly, they are classified into non-ML-based and ML-based approaches, as illustrated in Figure 4.1.

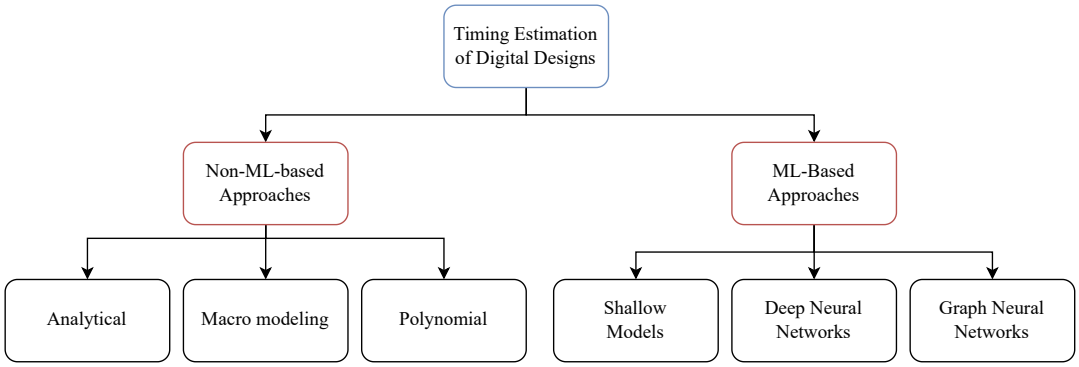


Figure 4.1.: Classification of related work to timing estimation of digital designs with and without using ML models.

4.2.1. Non-ML-based Approaches

The component delay estimation resembles the task of approximating a complex function. Classical methods, such as analytical, macro modeling, and polynomial-based approaches, are applied to solve such tasks.

4.2.1.1. Analytical

Analytical methods find an exact formula to describe the function being approximated. In this case, they relate a gate's delay to other design features based on the underlying physical and mathematical models.

The *logical effort (le)* method [118] relates the delay of a CMOS circuit to the *le* value. The latter is defined as the ratio of the input capacitance of a gate, and the input capacitance of an inverter delivering the same output current. The *le* value is estimated by counting transistor widths for the gate divided by transistor widths for the inverter. The gate delay is the sum of the zero-load delay and the *le* multiplied by the fan-out of the gate. The *le* method allows for comparing different implementations without running synthesis; it is technology-independent and can be used in multi-stage designs.

Wallace et al. [119] present a simple model to estimate the delay of a combinatorial cloud before its mapping to a gate-level netlist. They model the delay through each component on the cloud as a logarithmic function, dependent on the complexity of the component and its fan-out. The type of component and fan-out values are extracted from the designs before logic synthesis. These are scaled by constant coefficients or technology-dependent factors calculated using the NOT and NAND gate's delay, fan-out limit, and standard load. The authors show the efficacy of these methods in extracting parameters for three CMOS ASIC libraries and three industrial designs. However, the authors did not consider the effect of the signal transition times along the timing paths.

Qiu et al. [120] introduce four analytical methods to estimate post-routing delays given the

placement of a circuit. Using the same technology and router, sample designs are used to extract factors such as net degree, route length, net delay, and route Steiner tree length. Those factors are coefficients of the estimation functions, which can be quadratic or linear equations. For instance, the proposed *delay sampling* method computes delay as a quadratic function of the pin-to-pin placement length. The coefficients are computed with sample designs, and for new designs, the equation is used with the current wire's length. This method uses scaling factors and LUTs to link the estimated variables per net. Even though they report an average estimation error of 16%, this method can be slow and computationally expensive for larger designs.

Xu et al. [121] approximate the delay as a linear function of the number of programmable interconnect points and segments between two circuit points. Scaling coefficients are computed from observations done using the Xilinx® timing analysis tool. The approximation function estimates component delays in FPGA platforms, considering wiring effects, but it does not support ASICs.

4.2.1.2. Macro Modeling

Macro modeling methods build an abstract model of the function to approximate and decompose it with smaller blocks that can be approximated more easily. Macro modeling usually estimates design metrics from high-level circuit descriptions. It can be seen as a template in which the design metric under analysis is linked to a set of parameters and coefficients. This set of parameters is computed by using a set of sample designs.

Macii et al. [122] use a macro modeling technique to estimate the delay at the behavioral level. Technology-independent gate-level netlists are mapped to binary decision diagrams. For delay estimation, the model relates various types of the complexity of a Boolean function to its gate-level implementation, such as size, number of inputs, and density. Koyagi et al. [123] propose a macro model for delay estimation at RTL as a function of V_{dd} and V_t. The two scaling coefficients are extracted from experiments in HSPICE®. However, the model only supports NAND, NOR, and NOT gates.

4.2.1.3. Polynomial

Polynomial models involve approximating the function using a polynomial equation. They are regression-based, where the delay is the dependent variable, and design parameters are the independent variables.

QUEST [124] is a tool that uses *best-fit* polynomial estimation for area and delay from RTL descriptions. QUEST transforms the RTL code to a technology-independent form, similar to a logic synthesis tool. However, it only targets three logic gates: AND, OR, and NOT. The initial parameter extraction for the best-fit polynomial uses a few test designs. QUEST reaches a 5% error in delay estimation when testing with one hundred designs and four different technology libraries. Nemani et al. [125] use a one-shot scheme to predict the area-delay curve for Boolean

functions. This method proposes a novel *delay complexity* measure based on the *on and off-sets* of a Boolean function. According to the authors, the size and the number of prime implicants of a function affect the time needed to evaluate the function. An STA tool is used to generate a family of curves of delay values versus the novel complexity measure. This method provides a minimum and maximum delay estimation. However, it does not support circuits with XOR and other complex gates.

4.2.2. ML-based Approaches

Classical methods listed in Section 4.2.1 estimating delay at early design stages are constrained to the type of components or computationally expensive for larger designs. In contrast, ML-based models can learn patterns in the data without explicit knowledge of the mathematical or physical model. ML-based models are data-driven, i.e., they are trained on large datasets and can learn linear or non-linear relationships between data features and ground-truth labels. ML-based approximation functions are also faster during inference, can scale, and are a suitable approach when data are available, and the function to approximate is complex. Therefore, this thesis centers its attention on ML models used as function approximators for the timing of digital designs.

Figure 4.2 shows a high-level representation of how ML is used for design metric prediction in the digital design flow. The image illustrates the training and inference flows. Moreover, it shows that training circuits are passed through some stages of the design flow, from where the features are gathered. This is called the *input stage* and involves one or more stages of the design flow. Thus, it involves running different EDA tools for a given technology and hardware platform. The same input stage setup should be used over unseen circuits in the inference flow. A further stage in the design flow is executed to collect the ground-truth labels or objectives. This is called the *target prediction stage*. Usually, the target stage is a design stage that is costly in terms of computational resources or runtime, such as routing. Innovation in this ML-based flow happens by exploring different input and prediction stages with different features and objectives, data structures, and ML models.

This thesis focuses on applications using ML models for timing estimation at RTL. Shallow and deep ML methods have been applied to estimate pin-to-pin delays or timing metrics such as ATs or slacks. Usually, training circuits are mapped to Euclidean data, such as vectors or matrices, without considering that EDA objects are represented naturally as graphs [126], [127]. This mapping loses valuable information about the circuit's connectivity. The GNNs framework is an opportunity to solve EDA problems directly using graph structures, e.g., for intermediate RTL, gate-level, or transistor netlists. Therefore, this thesis classifies the ML-based applications to timing estimation based on the type of NNs used for Euclidean or graph-structured data.

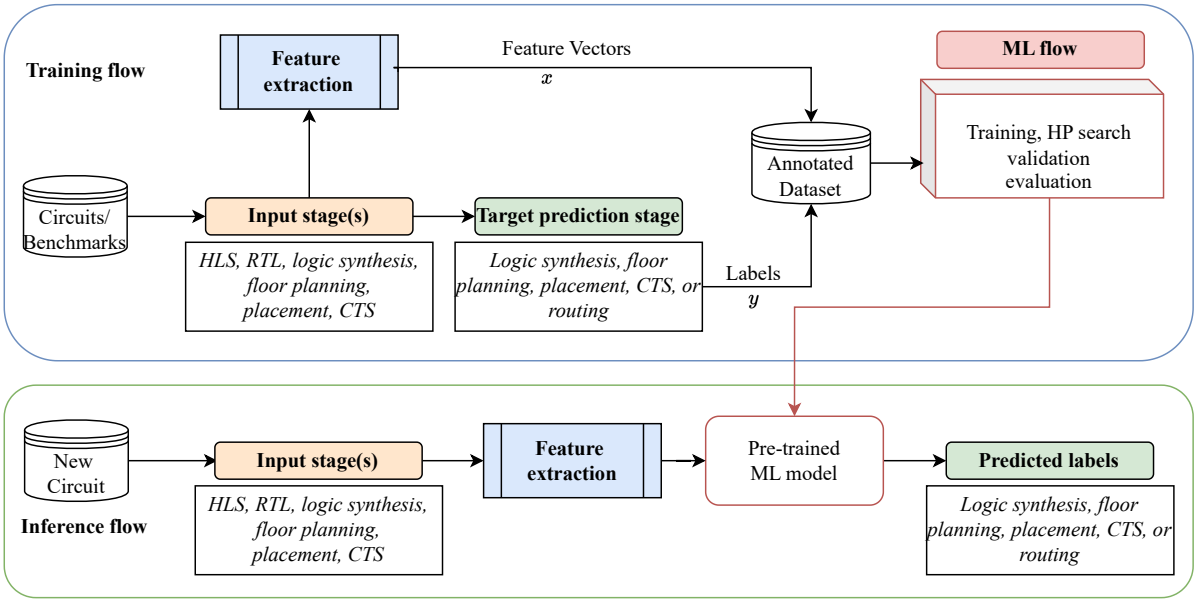


Figure 4.2.: Training and inference flows of ML models predicting digital design metrics.

4.2.2.1. Shallow Methods and Deep Neural Networks

ML Euclidean approaches have been used to estimate delays or ATs of circuit components at the early stages of the design flow, such as HLS [109], logic synthesis [128], and placement [129], [130].

The Pyramid framework proposed in [109] uses ML to estimate the optimal post-route target clock frequency and utilization of a design during HLS. Pyramid models the timing estimation as a regression problem, where the input features come directly from the HLS reports, and the objectives to predict are the maximum clock frequency, throughput, and area. Pyramid creates C-to-FPGA metrics for benchmark circuits and trains ML models such as a Random Forest (RF), Support Vector Machine (SVM), and an NN.

Ramesh et al. [128] use NNs to predict a circuit's AT after performing STA and Statistical Static Timing Analysis (SSTA). The NN is trained with results from commercial tools for ASIC synthesis and timing analysis. To that end, open-source benchmarks are used, and features, such as the number of inputs and outputs, gate count, number of nets, dynamic power, and gates in the critical path, are selected. The predicted ATs are further compared to the results of STA and SSTA.

Martin et al. [129] propose an ML-based net delay estimator for timing-driven FPGA placement. They use ML for three main reasons. First, training, validation, and inference are faster than traditional EDA tools. Delays are non-linear functions for which ML has proved to perform well. Finally, an ML-based estimator can be easily updated by expanding the training data distribution. They experiment with an MLP network, a Gradient Boost Regressor (GBR), and an RF as regression models. Those models are trained with 327 benchmarks based on a

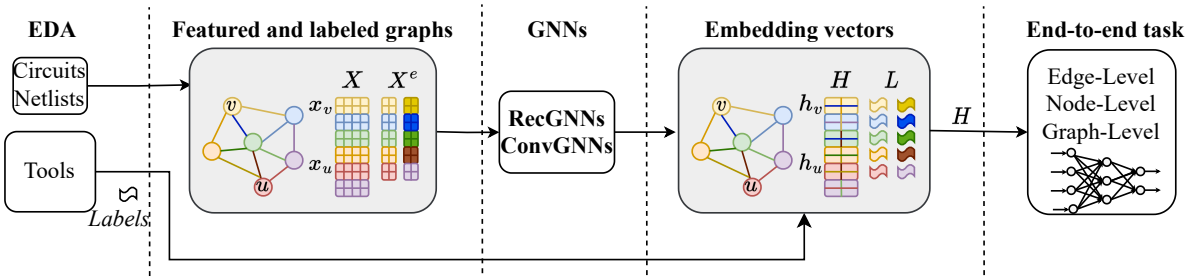


Figure 4.3.: General pipeline when using GNNs.

20 nm technology. The training dataset is built by extracting 20 features for each benchmark, where the most relevant are the number of pins, the half-perimeter wire length of the longest path, and the average congestion.

Barboza et al. [130] predict post-route net delays, slews, and slacks of netlists after placement. The models are trained using open-source benchmarks and a 45 nm technology library. The selected features include driver and sink capacitance, distance between target and sink node, max driver input slew, and context sink locations. They explore three ML-based regression models: Lasso, an RF, and a NN. The latter is trained to minimize an asymmetric MSE loss by penalizing underestimations.

Sengupta et al. [131] also ML to classify the timing paths of any RTL into different categories of path delay ranges. Moreover, they predict the worst-case delay and report all behavioral nodes in the worst-case delay path. This information allows for pinpointing critical components of the RTL and optimizing the design prior to synthesis, which is 40 times faster than the conventional flow.

4.2.2.2. Graph Neural Networks

GNNs [71] have already been employed in many stages of the digital design flow. Figure 4.3 shows the end-to-end flow when solving an EDA task using GNNs. In this case, EDA objects are used as graphs, i.e., feature and ground-truth labels from EDA tools are annotated to a graph structure. A GNN can extract node, edge, or graph embeddings, which are inputs to Euclidean ML methods that perform classification or regression at different levels. In particular, GNNs have shown superiority in design metric prediction. For instance, estimating net length [98] and routing congestion [101]. The computation flow of STA, building a timing graph, and propagating maximum delays along the paths resemble problems GNNs address. GNNs perform ML tasks by aggregating features and topology information from neighborhoods in input graphs. Therefore, recent approaches map design netlists to featured graphs and encode them using GNNs [132]. The resulting encoding vectors are used to estimate timing-related metrics, including post-routing ATs [133] and TNS [20], [134]. Inspired by conventional STA, some approaches estimate component delays and slews and then propagate them through the timing paths [135], [136].

Shrestha et al. [133] highlight the benefit of an early timing profile to avoid corrective circuit changes later in the design flow. They estimate the post-routing AT of a design's timing paths. To consider prediction results in three different scenarios, they take the input netlists after three stages: floor planning, placement, and CTS. They use open-source design benchmarks and a 65 nm technology to generate around 1500 physical layouts, from which timing paths are extracted and used for training. Results show that the prediction improves the closer the input stage is to the objective stage.

Lu et al. [134] propose an end-to-end framework to predict post-route TNS and detect failed runs earlier in the flow. The authors use GNNs to encode the netlists after detailed and global placement and CTS. The netlists from three sequential stages of the design flow are mapped to featured graphs and encoded by a global GNN. The node embeddings are then used as inputs to an MLP predicting the TNS per stage. This prediction achieves, on average, a 12.6% root-mean-squared error over seven industrial designs. Their final goal is to predict whether a design will meet the timing sign-off constraints, so they map the physical synthesis process to a sequential flow. Thus, the outputs of the GNN for the three stages are considered time series and are processed using an LSTM architecture. The GNN-based Long Short-Term Memory (LSTM) prediction achieves an error of less than 5.2%.

Ye et al. [135] present a GNN for path-based timing analysis. They propose a three-step flow: GNN encoding, cell slew as well as delay prediction, and critical-path AT calculation. Their GNN encoding can handle multidimensional edge features. The resulting encoding vectors are inputs to an MLP model, estimating the path-based cell slews and delays. The AT is the cumulative addition of the predicted cell delays and the wire delays obtained through graph-based analysis.

Guo et al. [136] present an end-to-end timing prediction engine. They map pre-routing netlists to Directed Acyclic Graphs (DAGs) and estimate post-routing timing metrics such as ATs and slacks at the design's endpoints. Their model targets two tasks: the cell delay as well as the slew calculation, and the AT as well as the slack propagation. The complete model is jointly trained by accumulating the losses of each task.

Approaches in [133], [134], [136] and [135] aim to optimize designs during physical synthesis. Thus, they build input graphs from pre-routing gate-level netlists. Work described in [136] and [135] have higher-quality predictions, i.e., higher R^2 scores, since they use more detailed gate-level netlists after placement as inputs. As shown in [134] and [133], the stage from which features are extracted affects the prediction results, i.e., features from post-placement netlists contain richer patterns than netlists after floor planning. However, the design's quality after physical synthesis depends on the quality of the initial RTL [137]. Thus, a timing-driven microarchitecture search is required at the RTL design stage. During the curse of this thesis, Sengupta et al. [20] map the Verilog RTL to an AST and use ML (shallow, deep, and GNN-based) models to predict the dynamic-power-performance Pareto front of an RTL. In this way, different microarchitectures and coding styles can be compared. Their approach reaches not only 95% prediction accuracy but is six orders of magnitude faster than running synthesis, power, and timing analysis tools.

Table 4.2.: Related work applying ML to the estimation of timing-related design metrics.

Cat.	Type	Ref.	Method	Estimated Metric	Input Stage	Predictions Stage	Hardware Platform
Non-ML-based	Analytical	[119]	Logarithmic functions	Component delay	RTL	Logic synthesis	ASIC
		[118]	Logical effort	Gate delay	MOS Circuit	MOS Circuit	ASIC
		[120]	Quadratic and linear equations	Delay	Placement	Routing	ASIC
		[121]	Linear functions	Component delay, max. path AT and area	HLS	Routing	FPGA
	Macro modeling	[122]	Binary decision diagrams	Component delay	Pre-RTL	Logic synthesis	ASIC
		[123]	Linear functions	Component delay	RTL	Logic synthesis	ASIC
	Polynomial	[124]	Best-fit polynomial	Path delays	RTL	Logic synthesis	ASIC
		[125]	Boolean functions	Min and max. combinatorial path delays	RTL	Logic synthesis	ASIC
ML-based	Euclidean Data	[109]	RF, SVM, NNs	Optimal clock frequency	HLS	Routing	FPGA
		[129]	RF, GBR, NNs	Net delay	Placement	Routing	FPGA
		[130]	RF, Lasso, NNs	Net delay and slew and endpoints slacks	Placement	Routing	ASIC
		[128]	NNs	AT after STA and SSTA	Gate-level	Routing	ASIC
		[131]	XGBoost	Path delays, worst delay	RTL	Placement	ASIC
	Graph Data	[133]	GNNs	AT of each timing path	Floorplan, placement CTS	Routing	ASIC
		[134]	GNNs LSTM	TNS	Detailed and global placement CTS	Routing	ASIC
		[135]	GNNs	Gate delay, slew and path AT	Placement	Routing	ASIC
		[136]	GNNs	Cell delay and slew, and ATs and slacks at endpoints	Placement	Routing	ASIC
		[20]	GNNs, RFs, NNs, Linear regressor	TNS and power	RTL	Placement	ASIC
		This thesis	GNNs, NNs	Component delay and slew, and block-based ATs, RATs, and slacks	RTL	Logic synthesis	ASIC

4.3. Summary and Comparison

Table 4.1 summarizes the visited work studying ML techniques to solve EDA tasks across the different stages of the design, such as design metric prediction or optimization with RL.

As mentioned in [134] and [133], the closer the input stage to the prediction stage, the better the quality of the predictions. Moreover, the closer the predictions are to the routing stage, the more confident and meaningful they are regarding sign-off closure. Therefore, many ML applications focus on physical design stages. However, the lower the abstraction level, the smaller the room for optimization and the lesser the benefits from early predictions. On the other hand, predictions from abstract specifications are very constrained, as their data, features, and models are limited to a specific IP. Similarly, HLS is an easy and fast approach from specifications to RTL design, but the high abstraction limits the flexibility and control over the design. This explains why this thesis focuses on predicting design metrics at the RTL design stage.

Focusing only on timing estimation at RTL, Table 4.2 summarizes the reviewed work and their most relevant features. In its last row, the methods of this thesis are also shown to provide a better understanding of the context and the contribution of this thesis to the state of the art. Similar to non-ML-based approaches, this thesis recognizes the need for a timing estimation at early stages of the design. Classical approaches are limited to a subset of component types or sample designs. Even though they are primarily technology-independent (i.e., they can be easily applied to different CMOS ASIC technologies), classical methods do not take advantage of the vast amount of data provided by EDA tools, i.e., they are not data-driven. ML approaches, on the other hand, generate a training set targeting one technology and are technology dependent. This thesis leverages MLP networks, GNNs, and RFs to estimate components delay, slew, and maximum ATs. Furthermore, using estimated delays, a block-based STA is conducted to obtain ATs, RATs, and slacks per each component in an RTL IR.

Table 4.3.: Major differences between this thesis and similar ML-based approaches.

Feature	[135]	[136]	[20]	[131]	**This thesis**
Input stage	Gate-level	Gate-level	RTL	RTL	RTL
Output stage	Routing	Routing	Placement	Placement	Logic synthesis
Leveraging RTL IRs from RTL generation tools	✗	✗	✗	✗	✓
Unconstrained number of training/evaluation designs	✗	✗	✗	✗	✓
Estimating path metrics with ML	✗	✓	✓	✓	✓
Propagating component delays	✓	✗	✗	✗	✓
Consideration of pin slews	✓	✓	✗	✗	✓
STA-like flow	✓	✓	✗	✗	✓
Pinpointing critical components	✗	✗	✗	✓	✓

This thesis is, from the targeted domain, comparable to [20], [135], [136] and [131]. Table 4.3 shows the significant differences between those approaches and this thesis. As with most ML approaches, the work in [135] and [136] take input features from stages where the RTL design is already fixed and mapped to a gate-level netlist, such as placement and CTS. They predict timing results after routing. Even though their predictions have higher quality and are closer to sign-off results, they use gate-level netlists as inputs, thus losing the mapping to the RTL.

Similar to this work, Sengupta et al. [20] and [131] recognize the importance of the RTL design and the impact of different microarchitectures and coding styles chosen. However, [20] focuses on estimating global design metrics such as TNS and do not analyze individual paths. They compare different microarchitectures at higher levels of abstraction but cannot **pinpoint critical components in the design**. Recent work [131] overcomes this by predicting the delay of all timing paths in an RTL. As a result, the worst-case path and its behavioral nodes are identified.

Even though work in [20] and [131] pursue the same objective of predicting timing-related metrics at RTL to enable optimization, there are some substantial differences. First, they build their training datasets by converting Verilog-based RTL to AST or intermediate representations, from which features are extracted. In contrast, this thesis recognizes that such representations are the middle stage within state-of-the-art RTL generation tools. Instead of returning from Verilog code to an AST, this thesis leverages RTL IRs from RTL generation frameworks. This way, going from specifications to an AST and finally to the Verilog code becomes an intuitive and automatic step. This not only avoids the usage of extra tools, such as Verilator [138], but allows automatically generating an unconstrained number of training and testing circuits. Consequently, the models in this thesis are trained with thousands of designs while previous work are constrained to a few benchmarks.

This thesis estimates the worst-case ATs using ML models directly, i.e., the ground-truth labels give information about the critical ATs at a design's endpoints. However, the limits of estimating path metrics with ML models is recognized, confining their solution space to the data distribution seen during training. This limitation is implicitly recognized in [131], where ML models classify path delays into bins instead of solving a regression task. This thesis recognizes that timing estimation at RTL itself depends on many factors, such as technology nodes, PVT conditions, synthesis tool parameters, and timing constraints. To train a model to directly predict the ATs of a path implies constraining the ML models and the solution space even more.

Therefore, this thesis maps the delay of individual primitive components to a regression problem and propagates predicted values to compute the conventional STA flow. The use of the RTL generation tool enables the consideration of individual components, which allows the modeling of the timing behavior of each component. Training designs are generated by targeting each component type, and pin-to-pin connections can be analyzed. This eases the consideration of the slew or transition time of the input signals. Moreover, analyzing RTL components instead of complete paths makes this approach generalizable, as target components are the primitive building blocks of any hardware IP.

This thesis presents a consolidated ML-enabled STA flow that leverages pin-to-pin delays and slews of individual components and propagates them. It borrows inspiration from approaches in [135] and [136] to compute path ATs and slacks in an STA-like flow. This thesis recognizes the ML's inability to completely generalize and solve the complex task of timing estimation at RTL. Instead, this thesis proposes the close integration of ML and classical techniques.

5. Data Generation Flow

Chapter 3 gives background concepts about ML methods and tools for solving regression tasks. In this thesis, the task to model is the STA flow of digital designs at the early stages of the design flow. To understand the nature of the problem, Chapter 2 explains the conventional flow with a focus on STA and RTL generation frameworks, which speed up the RTL design stage. The first step towards an ML flow is data collection. In contrast to other fields, EDA tools already generate massive amounts of data, and their results are assumed to be ground-truth labels. Thus, the data collection for EDA tasks is instead an automated data generation flow.

The digital design flow applies the *divide and conquer* strategy, producing a compendium of tools and teams working standalone. The disadvantages of this strategy are twofold: Iterations or corrective spins along the design flow are costly, as they involve different tools and teams. These generate a large number of data involving designs, netlists, layouts, and reports. Second, these EDA data are commonly used only by one specific team, focusing on analyzing a single design stage rather than the complete design flow. To overcome these disadvantages, the open-source toolchain OpenROAD [139] has been developed to reconnect standalone tools into one physical synthesis toolchain containing open-source designs, flows, and technologies. Moreover, OpenROAD is developing a data collection and storage infrastructure called METRICS 2.0 [140] to enable more ML applications for tool and flow outcomes prediction. Recognizing the importance of design stages at higher levels than the physical design, OpenROAD has not only connected its toolchain to Yosys [141] and ABC [117] but also to an RTL generation tool called Chisel [30], [142].

Drawing inspiration from OpenROAD, this work connects an RTL generation with a synthesis tool to enable ML applications. In particular, an in-house data collection flow is built as shown in Figure 5.1. The data collection process involves four components. First, MetaRTL converts a set of specifications to MoDs and Verilog-based RTL designs. The generated Verilog designs are mapped to gate-level netlists, which are then used for STA. The generated timing reports are preprocessed, and timing-relevant information is extracted, such as pin-to-pin delays or the ATs at the design endpoints. The latter are employed as ground-truth labels. Moreover, a *parser* extracts features from MoDs. These features are assembled with the labels by a *collector*, which builds the final either vector- or graph-based datasets. This thesis builds datasets covering diverse component types, technologies, and data structures. Therefore, the different explored parser and dataset collectors are explained in the following chapters. This chapter focuses only on the flow generating training circuits using MetaRTL and the synthesis flows to obtain their timing reports.

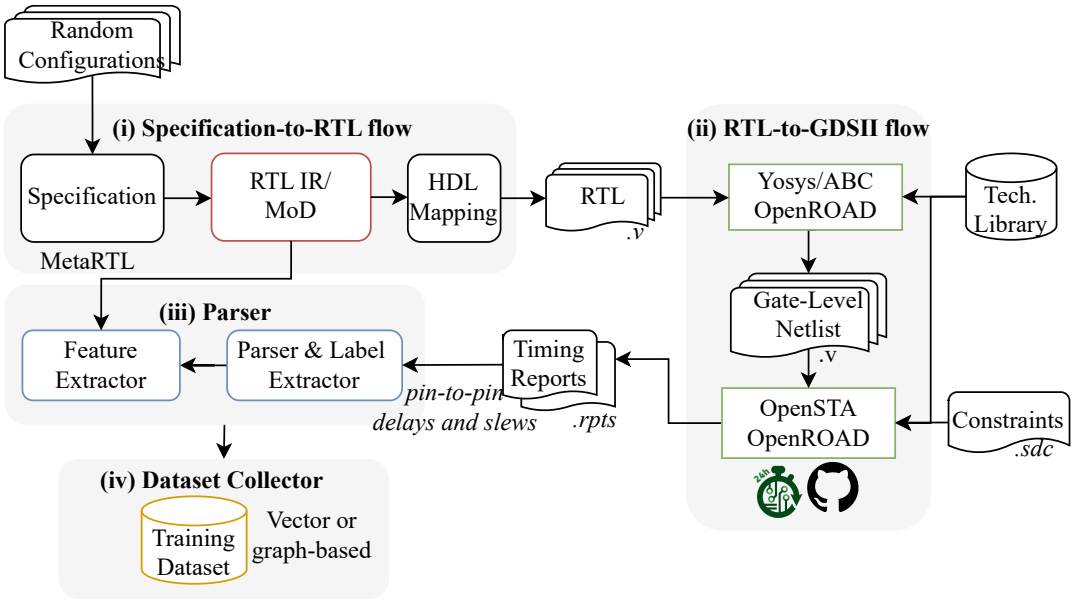


Figure 5.1.: Data collection flow.

5.1. Generating Training Circuits

This thesis uses MetaRTL, the Infineon in-house RTL generation framework to quickly and automatically generate training and evaluation circuits. The benefits of using MetaRTL are listed in Chapter 2. From the point of view of an ML application, MetaRTL allows easy data generation, automatically going from specifications to MoDs and Verilog designs and enabling feature engineering. Especially for the latter, the MoD is used as RTL IR, representing any design as a hierarchical tree composed of ports, connections, and primitive components. From this tree, the micro-architecture of the design and high-level features can be gathered to describe each one of its components, such as the bit-widths and the number of inputs and output pins. These granular component-level features allow for generalization to more complex designs, which are just the compendium of a more extensive set of primitive components. The detailed data generation flow in Figure 5.2 starts with generating random but valid design specifications for training circuits. Multiple training designs are created following four significant steps: generation of random specifications, ToDs, MoDs, and the corresponding Verilog.

5.1.1. Random Specifications

Random but valid and meaningful configurations are inputs to the flow in Figure 5.2. These are lists of dictionaries in JavaScript Object Notation (JSON) format specifying the attributes of each component within a design. These specifications target small circuits composed of different primitive components with different numbers of inputs, bit widths, and fan-outs. In order to build a robust dataset, different component types with different configurations are generated, e.g., one BAND with two, three, or N inputs, each with one, two, or N bits. Choosing these different configurations can be done in two ways: using *general* or exhaustive combinations and

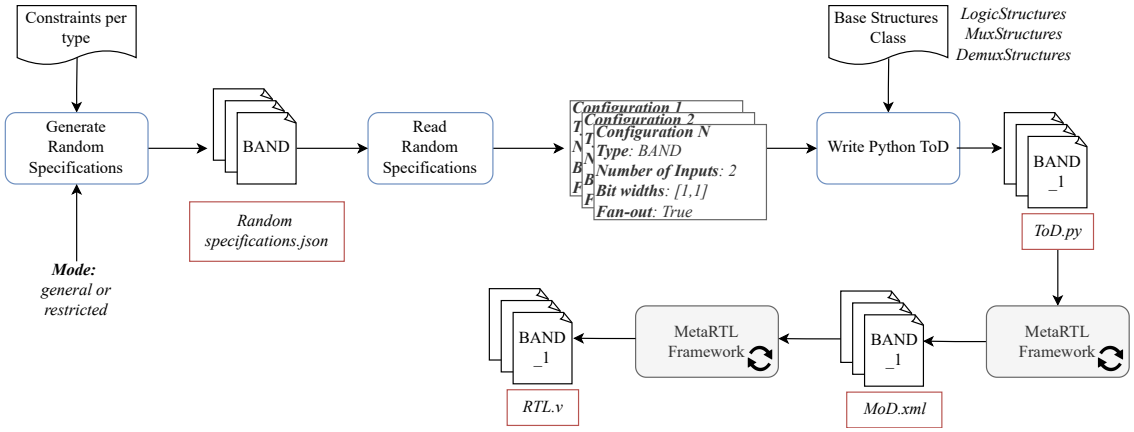


Figure 5.2.: Detailed specification-to-RTL flow.

restricted or selective specifications. Both rely on the definition of *valid* attributes, which relate to the definition of the primitive component given in Section 2.2.3.2. Valid attributes constrain the minimum and maximum values in which attributes are varied. For instance, knowing the maximum number of inputs supported for a DEMUX is two, specifying more than two input signals would be invalid. Similarly, specifying different bit widths for all inputs of a MUX or comparator is invalid but valid for certain logic operators.

Given the definition of valid attributes per type, the *general* specifications define the random dictionaries as all possible combinations of the number of inputs and bit widths. The general specifications vary the number of inputs according to Table 2.1 with a maximum value N of 32. Similarly, bit widths vary in a $[1, 32]$ range, and the number of connections to a primitive's output pin varies among $[0, 6]$. If there are zero connections to the output pin, the design would consist of a single primitive component.

The *restricted* specifications do not cover a defined range per component type, but particular configurations are picked from complex design IPs. Using a MetaRTL transformation, specifically a *model-to-text* transformation, information about an IP's sub-component types, number of inputs and outputs, their bit widths, and the number of connected components is printed and collected. These transformations are applied to over 140 IPs involving hardware adders, multipliers, RISC-V-based cores, and SoCs. The extracted information is used to determine the most common configurations per each primitive component. Afterward, all seen configurations per component type are saved and used as constraints for the random specification generation. Table A.2 lists the restricted specifications per component type, summarizing the number of inputs, their bit widths, and the fan-out. The number of outputs for most components is one. For the demultiplexers, the number of outputs changes from the range [1, 8] to 32, 64, 128, and 256.

Figure 5.3 shows a histogram per component type used over the 140 designs. The five most commonly used components are BAND, BXOR, BOR, and MUX. The histogram also shows other MetaRTL-supported component types not detailed in this thesis, such as lookup tables,

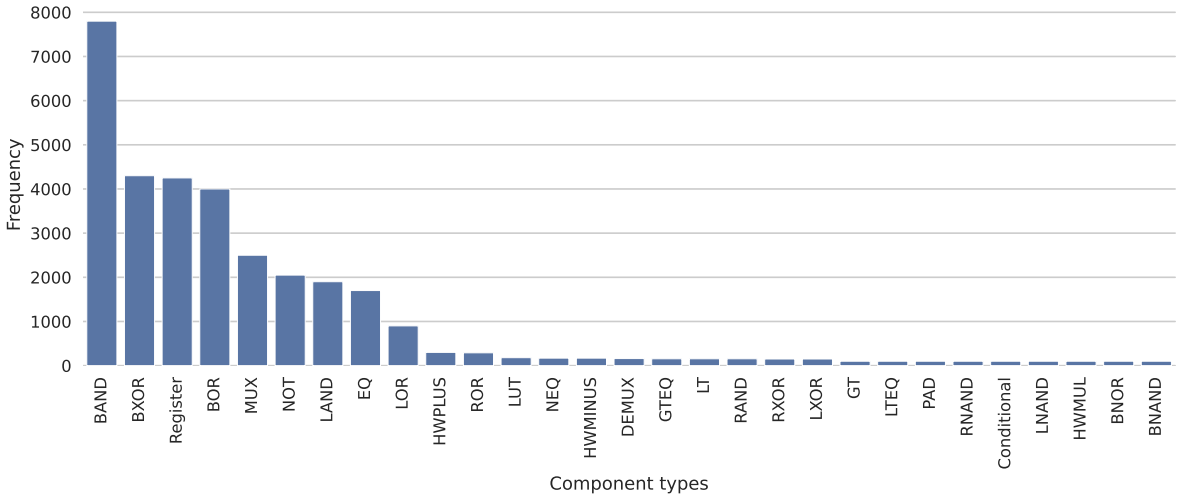


Figure 5.3.: Components used in existing hardware IPs ordered by type and frequency.

registers, and transcoders. These components are advanced primitives, and their configurations and timing behaviors are more complex than logic, arithmetic, or comparison operators. The restricted configurations are selected carefully, so the ones employed in actual designs are covered during training data. This avoids the generation of valid but not necessary configurations, especially the generation of training data for components that MetaRTL supports but never uses on practical IPs, such as the bitwise, reduced, and logical XNORs. During this thesis, different ML regression models are trained using restricted and general specifications. Further details concerning the supported component types and the total number of circuits are given per chapter.

5.1.2. Template of Design (ToDs)

Files containing random specifications are read using Python to build proper ToDs invoking MetaRTL libraries. A ToD is a Python function calling MetaRTL-based classes to instantiate ports, connections, submodules, and primitive components. One ToD must be generated per each random specification. To that end, the list of random specifications is traversed. For each specification, a Python script writes a ToD instantiating an object of the corresponding class.

The predefined primitive classes are written in Python and called *LogicStructures*, *MuxStructures*, and *DemuxStructures*. The former is used for arithmetic, comparators, and logic operators. These classes comprise three main steps. First, they generate the number of input ports with the specified bit widths. Depending on the target component type, the number of input and output ports differ; thus, special classes are used for MUXes and DEMUXes. Second, the output port and the main component are created from the specified type. Afterward, inputs and outputs are connected to the component. Finally, if there are extra components connected at the output pin of the main primitive component, these are instantiated and configured. All these four steps are realized using MetaRTL functions and classes. Moreover, they add all

Listing 2 Example of generated ToD Python file.

```
from base_classes import Logic_Structures

def main(Api, metaRTL):
   MoD = Logic_Structures(parent=metaRTL,
                          comp_type='BAND',
                          name='BAND_42', # Name of the current specification <type>_<id_number>
                          num_inputs=2, # Number of input pins
                          sizes=[3, 3], # Bit widths of input pins
                          fanout=2) # Number of connections to the output

   shell = MoD.create(metaRTL)
   metaRTL.insComponent(shell)
```

these artifacts to an MoD tree using the function *insComponent*. One example of the resulting Python ToD is shown in Listing 2. This ToD instantiates an MoD with BAND as the main component type, with two 3-bit input ports and two randomly chosen components at the output pin.

5.1.3. Model of Designs or RTL Intermediate Representations

The conversion from a ToD to an MoD is automated in MetaRTL, i.e., a MetaRTL command line is executed with the input ToD, and the output MoD is generated. This conversion step is executed per each generated ToD from the previous step. The resulting tree-like representation is exported in XML format. Listing 3 shows snippets of the resulting MoD generated using the ToD in Listing 2. As explained in Chapter 2, the MoD is a hierarchical tree of submodules, connections, ports, and components, each identified by a unique numerical identifier.

Listing 3 Example of a generated MoD XML file.

```
<MetaRTL>
    <Component type="Structure">
    <ID>1</ID>
    <Name>Top</Name>
    <Connections> ...
    </Connections>
    <Component type="BAND">
       <ID>5</ID>
       <Name>BAND_MoD_42</Name>
       <Port>
            <ID>2</ID>
            <Name>In00</Name>
            <Size>3</Size>
        </Port>
        ...
    </Component>
</MetaRTL>
```

XML files can be displayed as circuit design schematics. Figure 5.4 shows sample schematics of MoDs in the generated dataset. Figures 5.4a and 5.4b show the cases where the fan-out is zero, and the BAND and GTEQ components are targeted. A fan-out of two is used in the MoDs of Figures 5.4c and 5.4d. Both target the BAND as the main component type but with different numbers of inputs and bit widths. In particular, Figure 5.4c represents the MoD in Listing 3.

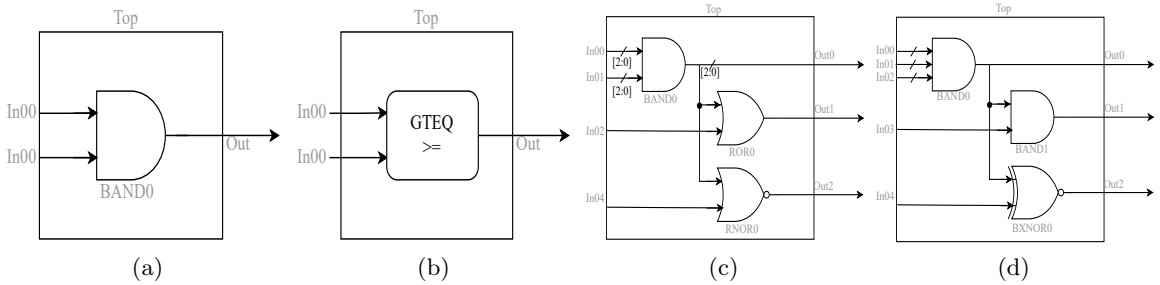


Figure 5.4.: Example of RTL IR or MoD in the training dataset.

5.1.4. Verilog-based HDL

Similarly, the conversion from MoD to Verilog is automated in MetaRTL, i.e., the output code is generated by executing a command with the input MoD and the target HDL language. In this way, per each random specification, a final Verilog file is generated. The generation steps are automated to traverse all component types and their random specifications. The flow is automatically run per each set of random configurations, generating the ToD, MoD, and HDL. However, only the latter two artifacts are later used in the ML flow. Listing 4 shows snippets of the resulting Verilog code for the MoD in Listing 3.

5.2. Generating Ground-Truth Labels

Figure 5.1 shows that the generated HDL is the input to the synthesis flow. The generated Verilog designs undergo synthesis and timing analysis, utilizing predefined tools, parameters, clocks, and timing constraints. The generated reports are collected, parsed, and processed to extract ground-truth labels. The following sections give practical details about the synthesis and timing analysis flows.

5.2.1. Obtaining Gate-Level Netlists

This thesis uses open-source synthesis tools, such as Yosys [141] and ABC [117], which are connected to the OpenROAD toolchain. Yosys is a Verilog synthesis tool that allows for the addition and customization of extensions. Nevertheless, it does not include a timing analysis engine, i.e., the logic synthesis is not timing-driven. Yosys uses ABC for logic minimization and technology mapping.

Logic synthesis can be performed hierarchically or flattened. The latter flattens the design so that only one top module is present. Flattening enables the tool for cross-boundary optimizations. However, RTL components, connections, and ports may be renamed, and thus, the mapping between the RTL and the gate-level netlist becomes cumbersome. Figure 5.5 shows a high-level example of the graph of a SoC design when considering hierarchies and flattening to one single module. The logic optimizations highlighted in blue are performed for each hierarchy or the flattened module. Logic synthesis is done hierarchically in this thesis to keep RTL

Listing 4 Example of a generated Verilog file.

```
module Top (
    input [2:0] In00,
    input [2:0] In01,
    input In02,
    input In03
    output [2:0] Out0,
    output Out1,
    output Out2,
);

wire   [2:0] BAND_MoD_42_Outp_s;
Top_BAND_MoD_42 comp_BAND_MoD_42(
    .In00 ( In00               ),
    .In01 ( In01               ),
    .Outp ( BAND_MoD_42_Outp_s )
);
Top_BAND_MoD_42_ROR comp_BAND_MoD_42_ROR(
    .In00 ( BAND_MoD_42_Outp_s ),
    .In01 ( In02         ),
    .Outp ( Out1        )
);
Top_BAND_MoD_42_RNOR comp_BAND_MoD_42_RNOR(
    .In00 ( BAND_MoD_42_Outp_s ),
    .In01 ( In03         ),
    .Outp ( Out2        )
);
assign Out0 = BAND_MoD_42_Outp_s;

endmodule
```

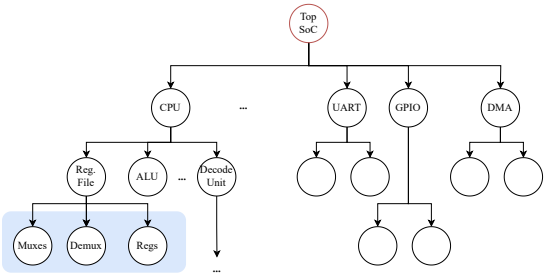
boundaries and component names in the gate-level netlist. This mapping allows identifying RTL components from timing reports and back-annotate delays, slews, and ATs accordingly.

The logic synthesis tool needs, besides the Verilog-based RTL, the technology library file [23] available in a PDK. Throughout the thesis, two different PDKs are used: the open-source SkyWater [143] 130 nm and a proprietary 40 nm. Table B.1 on Page 181 compares relevant aspects of both technology nodes.

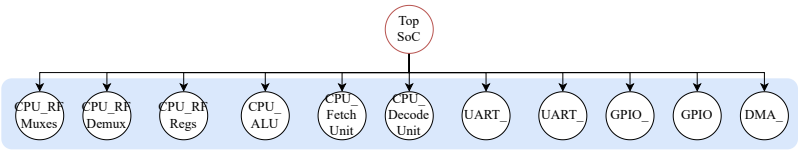
Finally, the proposed data collection flow does not include floor planning, placement, or routing. As suggested in related work, the closer the input design stage is to the prediction stage, the better results can be achieved. However, this thesis focuses on the RTL stage and targets the MoD as input, generalizing to any hardware design, HDL, or target platform. Moreover, the presented flow can be extended to go deeper into the physical design stages by running the open-source tools from the OpenROAD toolchain and gathering from there the ground-truth labels. Nevertheless, this would cause an increase in runtime for generating training data. Even though timing analysis after logic synthesis does not consider wire timing models or parasitic effects, it provides a good intuition about the initial timing behavior of the netlist.

5.2.2. Getting Timing Reports

The OpenSTA [144] tool from the OpenROAD toolchain is used for generating the timing reports. OpenSTA is a gate-level static timing checker that can be used to verify the timing of



(a) Hierarchical view



(b) Flattened view

Figure 5.5.: Views of components on a SoC. Blue boxes represent the logic optimizations.

Table 5.1.: STA configuration parameters when using the 40 nm technology.

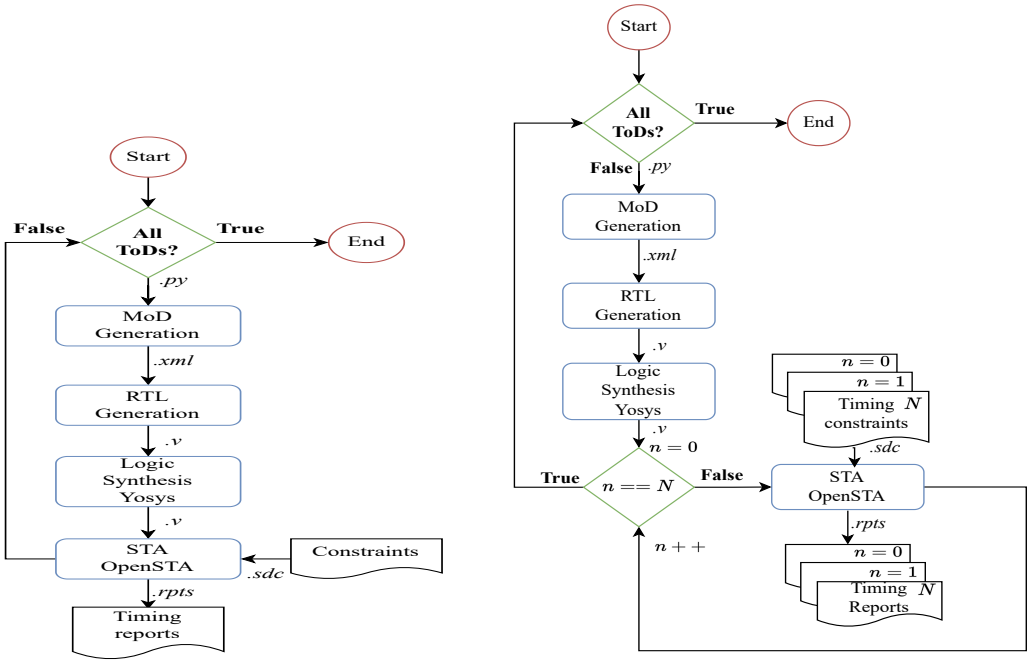
Setup	Name	Value	Unit
Ports	Input Delay	Varies*	ns
	Output Delay	Varies*	ns
	Input Load	0.0075	fF
	Output Load	0.015	fF
	Input Slew	Varies*	$ns[rise, fall]$
Clock	Period	6.6	ns
	Latency	[0.1, 1.0]	$[min, max]$
	Transition	[0.1, 0.2]	$ns[rise, fall]$
	Skew	[0.6, 0.15]	$[setup, hold]$

* Different selected ranges per dataset.

designs and needs as inputs a gate-level netlist, the technology library, and predefined timing constraints in SDC format. In the technology library, each gate's timing behavior or NLDMs are described for a specific corner case in terms of PVT values. The timing constraints in the SDC input file define the clock and port configuration. Table 5.1 lists key STA configuration parameters when using a 40 nm technology. As delays and slews are the values to estimate, different datasets are built varying the initial delays and slews for the primary ports. This allows the generation of a training dataset with a better distribution of these features and ground-truth labels. The particular values used will be explicitly mentioned for each dataset.

Two synthesis flow settings are explored to perform timing analysis with one or N different constraints simultaneously, as shown in Figure 5.6. The flow on the left can be used when running just a few different STA configurations with different values for the primary ports used. However, this flow becomes inefficient for N different values as logic synthesis is unnecessarily re-run for the same netlist N times. The intuitive flow on the right proposes to use the same netlist for N different configurations of primary ports. Each run can be executed parallelly with

separated timing constraints and generate separated reports. This speeds up the generation of timing reports per each netlist in the training circuits.



(a) One STA configuration (b) N parallel STA configurations

Figure 5.6.: Flows running one or parallel STA setups for the same netlist.

Figure 5.7 shows the detailed steps when running OpenSTA to obtain timing reports listing the delays and slews in our target component and pins. Using a *model-to-text* transformation in MetaRTL and a given MoD, the hierarchical name of each component and its pins are extracted and exported as a list of dictionaries in JSON format. This information is used to generate reports using Tool Command Language (TCL) and the command syntax proper of OpenSTA. A TCL script is automatically built to consider timing paths passing through the desired pin-to-pin connections. For each component in an MoD, a command line is written for the input-to-output timing path with path delays, i.e., maximum, and minimum rising and falling delays. All command lines are saved into a TCL file. Listing 5 exemplifies a generated command line in the TCL file for the MoD in Listing 3. Line 1 uses OpenSTA commands to generate timing reports and configure them to show particular fields. The pins *through* which the path must go are listed in Line 2. Line 3 specifies the path group and type. Finally, Line 4 redirects the generated report to a file with proper naming conventions.

The TCL file, the gate-level netlist, the technology library, and the SDC constraints, are inputs to OpenSTA. The execution of STA begins by setting up environment variables, such as file/directory paths. Afterward, the timing constraints from the SDC file are read and used to generate commands for clock and port configurations. Finally, the generated TCL file is executed, producing multiple timing reports. The benefit of the flow described in Figure 5.7 is that timing reports are automatically generated for any MoD, passing through all components'

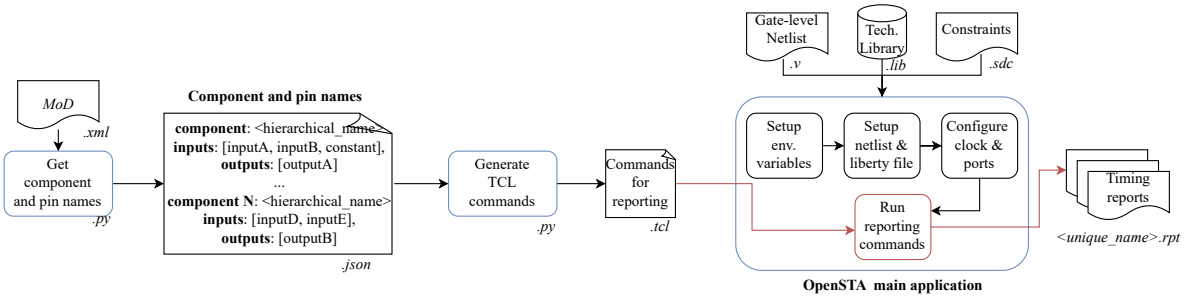


Figure 5.7.: Detailed OpenSTA flow.

pin-to-pin connections. This is suitable for dataset generation when hundreds of MoDs should be considered.

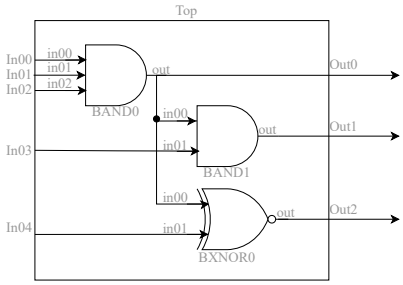
Listing 5 Example of a command line in a generated TCL file.

```
1 report_checks -no_line_split -digits 4 -fields {capacitance slew input_pins nets fanout}
2 -through comp_Top/comp_BAND_MoD_42/In00 -through comp_Top/comp_BAND_MoD_42/Out0
3 -path_group INOUT -path_delay max_rise
4 > Top__BAND_MoD_42__In00__Out0.INOUT.max_rise.rpt
```

In the next step, the generated timing reports are automatically read and parsed, and timing-relevant information, such as pin-to-pin slews, delays, and endpoints' ATs, is extracted. These serve as ground-truth labels for the ML models. Figure 5.8 shows an MoD and one of the generated timing reports. A timing report is composed of four main parts. First, the header specifies the path's start point, end point, type, and group. Afterward, the data path delay is described, giving information on the pin-to-pin delays for each gate in the path. The mapping of MoD components and gates in the netlist is only sometimes one-to-one. If one MoD component is mapped to several gates, the delay through all these gates is added and considered as the delay of the MoD component. The data arrival time accumulates all individual cell delays through the path. Finally, the timing report's third and fourth blocks show the calculation of the RAT and slack, respectively. It is important to note that the timing report in Figure 5.8b provides a specific description of the components using the hierarchical names of the MoD. This is only possible after hierarchical synthesis; otherwise, the components would be flattened, optimized, and renamed, thus losing their mapping to the RTL design.

5.3. Summary and Comparison with Other Tools

This thesis proposes the automated flow for data generation shown in Figure 5.1, which leverages an in-house RTL generation framework and open-source synthesis tools. This data generation flow generates random specifications, their corresponding MoDs and Verilog files. From the former, features describing the components and the overall design are extracted. Meanwhile the Verilog-based RTL is passed through hierarchical logic synthesis. The gate-level netlist is an input to OpenSTA, which executes commands generated and dumped to TCL scripts. These TCL files specify the desired timing reports and their specific configurations. Finally, timing reports are collected and parsed, and ground-truth labels are extracted.



(a) Sample MoD

```
Startpoint: In02 (input port clocked by clk)
Endpoint: Out2 (output port clocked by clk)
Path Group: INOUT
Path Type: max
Fanout       Cap      Slew     Delay      Time   Description
---------------------------------------------------------------------------------------
                     0.1000    0.0000    0.0000   clock clk (rise edge)
                               1.0000    1.0000   clock network delay (ideal)
                               0.4000    1.4000 v input external delay
                     0.2000    0.0000    1.4000 v In02 (in)
    1    0.0024                                   In02 (net)
                     0.2000    0.0000    1.4000 v comp_BAND0/_0_/C
                     0.0416    0.1771    1.5771 v comp_BAND0/_0_/X
         0.0070                                   Out0 (net)
                     0.0416    0.0000    1.5771 v comp_BXNOR0/_0_/B
                     0.0371    0.0847    1.6618 v comp_BXNOR0/_0_/Y
    1    0.0000                                   Out2 (net)
                     0.0371    0.0000    1.6618 v Out2 (out)
                               1.6618   data arrival time

                     0.1000    6.0000    6.0000   clock clk (rise edge)
                               1.0000    7.0000   clock network delay (ideal)
                              -0.6000    6.4000   clock uncertainty
                               0.0000    6.4000   clock reconvergence pessimism
                              -0.8000    5.6000   output external delay
                               5.6000   data required time
---------------------------------------------------------------------------------------
                               5.6000   data required time
                              -1.6618   data arrival time
---------------------------------------------------------------------------------------
                               3.9382   slack (MET)
```

(b) One generated timing report

Figure 5.8.: Example of an MoD and one of the automatically generated timing reports.

The flow in Figure 5.1 is compatible with any RTL generation framework using an AST representation as RTL IR. The AST allows to describe a design in terms of modules and components, which timing behavior need to be estimated. This thesis employs MetaRTL [32] as it is the Infineon standard in-house tool and provides the above-mentioned benefits. Also, commercial synthesis tools can be used to generate timing reports during the synthesis flow. Open-source tools are used, as they provide more transparency in the design flow and allow thousands of runs without incurring excessive license costs. Open-source tools enable multiple ML applications and research in the EDA field, even within industrial design flows, as described in Appendix B. Qualitative comparisons among open-source and commercial tools have been done in the literature [145]. Kahng et al. [145] list the features of each tool when having open and free software or counting with years of experience and billions of investments. Table 5.2 summarizes the highlighted main differences between the two types of EDA tools.

Table 5.2.: Qualitative comparison between open-source and proprietary EDA tools.

Type / **EDA Tool**	**Extensibility**	**Accessibility**	**Scalability**	**Customer support**	**Reliability**	**Technology & engineering**	**Workforce & development**
Open-source	✓	✓	✓				✓
Proprietary				✓	✓	✓	✓

Moreover, ML models are data-driven methods. Thus, the quality of their predictions correlates with the training data and ground-truth label quality. This thesis extracts ground-truth labels directly from timing reports generated by the open-source EDA tools. Therefore, a quantitative comparison is conducted to evaluate how close these ground-truth labels from open-source tools are to values generated by a commercial tool. Appendix B.2 describes the workflow to compare toolchains when synthesizing five different CPUs. Using MetaRISC [53], five RISC-V implementations are generated using different extensions. Timing, area, and power reports are collected using the same constraints and initializations.

Appendix B.2 describes experiments with varying tools, PDKs, and clock configurations. Results for the 40 nm PDK, flattened synthesis, and a clock period of 25 nm are shown in Table 5.3. Averaging the post-routing factors, the commercial tool outperforms OpenROAD by 2.5. Moreover, on average, the commercial tool is faster without parallelization and could meet the timing constraints for lower clock periods better than OpenROAD. However, this is expected to change based on the rapid development of OpenROAD and its community commitment [146]. In contrast to commercial tools, open-source EDA tools open the door to academic research, benchmarking, ML techniques, and the transfer of research to real-world scenarios. Thus, this thesis employs OpenROAD also to encourage open-source projects.

Table 5.3.: Quantitative comparison between open-source and proprietary EDA tools.

Stage	**Ratio [Open-source Tool/Proprietary Tool]**			
	NAND2 Eq. Area	**No. Standard Cells**	**Worst Slack (WS)**	**Total Power**
Post-logic synthesis	1.5	1.8	<0	2.4
Post-route	2.1	2.4	2.9	2.7

6. Pin-to-pin Component Delay and Slew Estimation

During STA, component delays and slews are approximated and propagated along the timing paths to calculate global design metrics as described in Chapter 2. Global metrics, such as ATs and slacks, indicate whether a design meets the timing constraints. Even though the process of designing an RTL, synthesizing it, and running STA tools may seem straightforward, it requires considerable time and computational resources. This becomes counterproductive when even minor modifications to the RTL design necessitate a re-iteration of the entire flow. In such cases, the benefits of estimating early-stage timing metrics become evident. This demands for faster and more accurate timing estimation has also been recognized in the state of the art as explained in Chapter 4.

The first step in estimating timing for an entire RTL design involves assessing individual components, i.e., analyzing how much time is required for a signal to go from the input to the output pin of every component in the whole design. At the gate level, this question is straightforward due to the availability of timing models per gate in the technology library files. However, at the RTL, the challenges for the estimation are twofold. Firstly, the Verilog-based RTL is an abstract representation of the data flow across hardware registers and logical operations, i.e., no concrete logic gates are implemented. To overcome this, RTL IRs are used to map the RTL design to a structural description using components and modules while being independent of the target HDL and hardware platform. Secondly, mapping components in RTL or RTL IR to gates in the target technology is not a straightforward one-to-one or unique process. Technology mapping is just one aspect of the logic synthesis tool's responsibilities; it also encompasses Boolean optimization, which influences how the mapping is executed.

Classical methods for estimating RTL component delays from a higher level of abstraction use polynomial, macro modeling, and analytical approaches. As discussed in Chapter 4, these methods have limitations such as being slow, constrained by component type, computationally expensive, or providing only a rough estimation. Conversely, a gap exists in ML methods for approximating the timing of RTL designs. This chapter introduces innovative methods using NNs for pin-to-pin component delay estimation of RTL components in ASIC designs. The following sections outline the estimation problem and the essential steps to address it using the ML methods from Chapter 3.

6.1. Problem Statement

Given an RTL IR comprising combinatorial logic components, estimations of the delays and slews over each pin-to-pin connection are required to evaluate the timing of the complete design. Pin-to-pin values serve to detect critical pins within individual components. For example, in

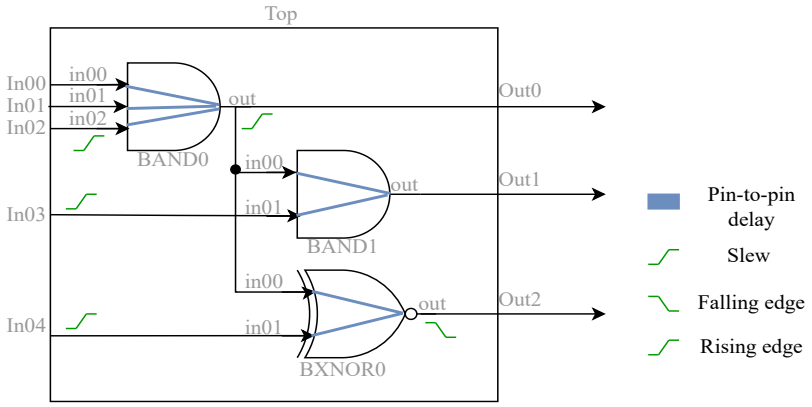


Figure 6.1.: RTL IR of a combinatorial circuit with pin-to-pin delays to estimate.

the schematic view depicted in Figure 6.1, it is imperative to predict delay and slew values for all the pin-to-pin connections highlighted in blue.

The timing estimation of RTL IRs is challenging, as it implicitly abstracts the complete flow from the generation of RTL to timing reports. Specifically, this abstraction approximates Boolean optimizations and technology mapping done by the logic synthesis tool, as well as the timing models described in the technology library. These timing models further abstract the physical behavior of the gates in the technology, which are influenced by PVT conditions, gate type, and pin-specific features, such as load and transition times. This thesis proposes ML models to estimate pin-to-pin delays and slews of primitive components in an RTL IR, replacing the RTL generation, logic synthesis, and STA tools with an ML-based flow. As highlighted in Figure 4.2, the main benefit of ML models lies in the inference flow, where early estimations are obtained without executing EDA tools, thus reducing runtime.

In this chapter, the delay and slew estimation is defined as a regression task. Regression models are trained using a supervised learning strategy to learn the relationship between data features and ground-truth labels. The input to a regression model is a feature vector $\mathbf{x_i}$, where i is a pin-to-pin connection of a component within a design. The vector $\mathbf{x}$ encodes information on the selected features describing the pin-to-pin connection. For instance, the RTL IR in Figure 6.1, which consists of seven pin-to-pin connections highlighted in blue, is represented by seven rows or samples in the dataset. The model's output $\mathbf{y_i}$ is a vector representing the regression objectives for each pin-to-pin connection i. This chapter uses delays d and output slews s^{OUT} as objectives and considers maximum and minimum delays, as well as rising r and falling f edges, denoted as $\mathbf{y_i} = [s_f^{OUT}, s_r^{OUT}, d_f, d_r]$.

As discussed in Chapter 3, solving a regression task with ML entails several essential steps, as illustrated in Figure 3.1. In essence, features are carefully selected and extracted from the training data. Subsequently, datasets containing features and objectives are constructed and divided for training, validation, and testing. Next, ML model architectures that align with the training data structure are selected. Afterward, the models' hyperparameters are fine-tuned, and the models undergo training. Finally, the trained models are evaluated using unseen data.

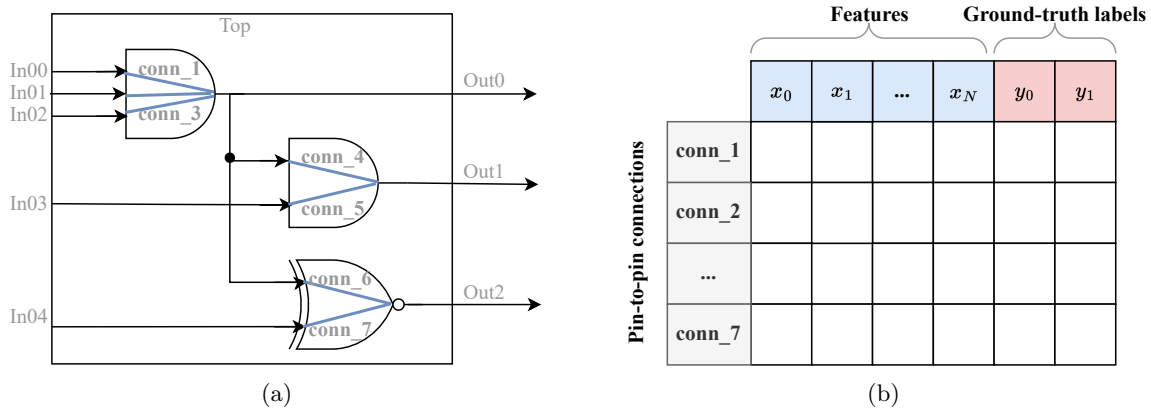


Figure 6.2.: From pin-to-pin connections in a MoD (left) to a tabular dataset (right).

The following sections delve into the implementation details of these steps as applied to the specific regression task of delay estimation.

6.2. Dataset Collection

Chapter 5 describes the data generation flow comprehensively, going from random design specifications to timing reports, from which ground-truth labels, i.e., pin-to-pin delays and slews, are extracted. As depicted in Figures 5.1 and 5.2, the generation of training circuits commences with the definition of the component types to be targeted in the dataset. The more component types, the more complex becomes the regression task. Thus, different subsets of component types are employed. Additionally, drawing inspiration from the reviewed state-of-the-art methods listed in Table 4.2 and applying ML to estimate timing-related metrics, both tabular and graph datasets are constructed. The flexibility of the training generation process facilitates the effortless construction of both datasets by simply adding a *model-to-text* transformation script. This approach has led to the generation of three distinct datasets:

6.2.1. Tabular Datasets

In a tabular dataset, each row represents a pin-to-pin connection of a logic component in an RTL IR. An illustration of this mapping is shown in Figure 6.2, where the seven pin-to-pin connections of the design are mapped to seven rows in a tabular format. Every connection is assigned to a row, while the columns exhibit the selected features $\mathbf{x}$ and objectives $\mathbf{y}$. Two different tabular training sets are generated: A basic dataset (*DS-I*) and a generalized dataset (*DS-II*). To consider variations of the slew, timing reports with varying variants for the slews of the input ports are generated. More precisely, two and four different values are used for the datasets DS-I and DS-II, respectively. In both cases, the synthesis flow in Figure 5.6a is employed. The resulting quantity of circuits and samples (rows) for each component type within each tabular dataset is enumerated in Table 6.1.

Table 6.1.: Component types covered in tabular datasets DS-I and DS-II.

Type	Type	No. Circuits	No. Samples
Logical[1,2]	AND, NAND, OR, NOR, XOR, XNOR, NOT	200.9K	2564K
Arithmetic	CMULT, CDIV, CMOD, CREM, CPLUS, CMINUS, HWPLUS, HWMINUS, HWMUL	68.6K	665k
Multiplexing	MUX[2], DEMUX	1.0K	78K
Comparators	GT, GTEQ, LT, LTEQ, EQ, NEQ	5.5K	126K

[1] Considering three fashions: logical, reduced, and bitwise.
[2] Component type considered in DS-I.

6.2.1.1. Dataset I (DS-I)

The training circuits used in this dataset contain twenty types of RTL IR components, including MUXes, inverters, and various types of logic gates (AND, NAND, OR, NOR, XOR, and XNOR) that are used in a bitwise, logical, and reduced manner. Within the DS-I dataset, each training design comprises a randomly generated number of inputs and bit widths, with the number of inputs ranging from one to twelve and the bit widths ranging from one to thirty-two. The number of components per design varies from zero to six to account for different fan-outs, and single-component designs have no connected components to the output pin. In addition, two different transition times for the primary input ports are considered during the STA. Overall, DS-I comprises 42.9K training samples.

6.2.1.2. Dataset II (DS-II)

To expand the coverage of the dataset, DS-I is now enlarged to encompass other component types, such as DEMUXes, arithmetic operators, and comparators. This enhanced dataset is called DS-II and includes 37 types of primitive components listed in Table 6.1. The input bit widths range from one to 32 for up to 32 inputs. Different component types have different ranges for the selected RTL features. For instance, a comparator such as the GT has two inputs and one output, while for logic gates, the number of inputs is at least one. Thus, in the generated random configurations, the number of inputs for GT is fixed to two, while for logic gates, it is in the $[1, 32]$ range. This results in an imbalanced training dataset with respect to the component type, as shown in Table 6.1, where the number of circuits using logical gates is 36.5× larger than the number of circuits for comparators. In total, the collected dataset DS-II consists of 3.4M samples.

6.2.2. Graph Dataset

Using graph structures, a third dataset, *DS-III*, is built. The training circuits used for *DS-III* contain up to six basic components. When the number of components exceeds one, the first gate output is the input to the other gates. The number of inputs to a component varies from one to eight. Similarly, each input's bit widths vary between 1, 2, 4, and 8-bit. Finally, inverters

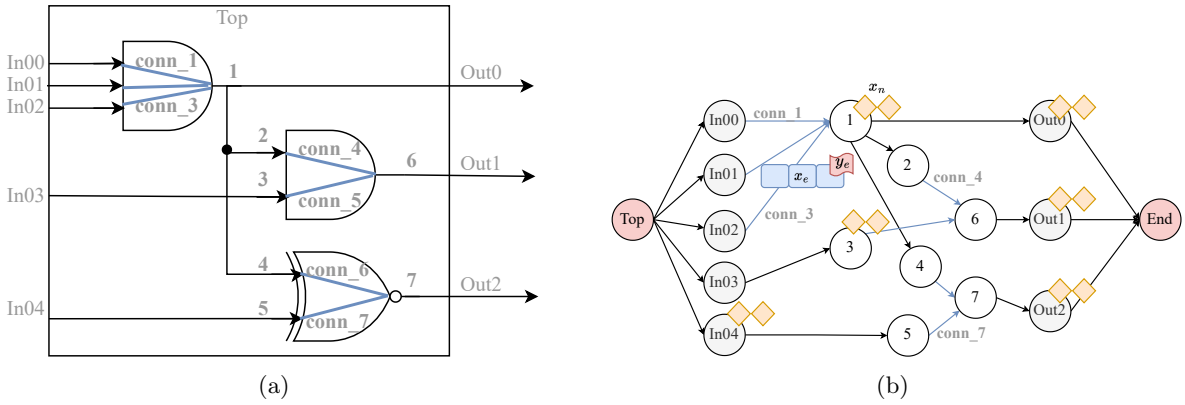


Figure 6.3.: From pin-to-pin connections in a MoD (left) to a graph (right).

(NOT) and logic gates are the component used. The input transition times or slews of the primary input ports vary to cover four corner points. For both rising and falling edges, the four $[min, max]$ tuples considered are $[0, 0]$, $[0.5, 0, 5]$, $[1.0, 1.0]$, and $[0.0, 1.0]$. The total number of generated designs varies depending on the component type: 803 circuits are generated for each logical and reduced gate, for each bitwise gate 774, and for the NOT gate 106. Additionally, running STA for the four input transition times produces a final dataset of 56336 graphs.

In the graph-based dataset, instead of mapping pin-to-pin connections to samples or rows of a table, an RTL IR is mapped to a featured DAG, which is annotated with features per node and edge. These DAGs are a particular type of graph defined in Section 3.1.4.3.1. This thesis maps designs to graphs using the pin node convention, i.e., the pins of the design are the nodes of the graph, and their connections are the edges, as illustrated in Figure 6.3. Edges in blue correspond to the pin-to-pin connections, which delays and slews are depicted with a red flag. These are the estimation goals of this chapter. If the edge connects an output pin with the input pin of another component or sub-module in the hierarchy, it is considered an *ideal wire* with zero delay.

6.3. Feature Selection and Analysis

After generating training circuits, meaningful features from the data are selected to match the nature of the task. As explained in Chapter 3, features should be selected, analyzed, and preprocessed to be numerical, and the data types expected by the model should be used.

As the ground-truth labels are extracted from STA reports, we draw inspiration from the Non-Linear Delay Models (NLDMs) used in STA tools to determine the training features. As explained in Chapter 3, these models define the time or delay between a gate's input and output pins using LUTs. These are indexed by the fan-out load and slew at the input pin, as depicted in Figure 2.11. Consequently, the chosen features for each pin-to-pin connection relate to factors like capacitance, bit widths, number of component pins, and current pin connection. These features are taken from the RTL IR, i.e., the input to the proposed ML models is at a higher abstraction level than a gate-level netlist.

Table 6.2.: Features and objectives of tabular datasets DS-I and DS-II.

Features & Objectives	Symbol	Size	Type	Description
Component type	x_0	T	`bin`	One of T[1] possible types
Input number	x_1	1	`int`	Current input pin considered
No. Inputs	x_2	1	`int`	Random number $\in [1, B]$[2]
Bit width input	x_3	1	`int`	Random number $\in [1, 32]$
Bit width output	x_4	1	`int`	Random number $\in [1, 32]$
Connections to input	x_5	1	`int`	Connected components $\in [0, 6]$
Connections to output	x_6	1	`int`	Connected components $\in [0, 6]$
Input slew	x_7	4	`float`	Transition times for current pin
No. Outputs[3]	x_8	1	`int`	Random number $\in [1, 32]$
Sum bit width inputs[3]	x_9	1	`int`	Sum over all inputs bit width
Avg. bit width inputs[3]	x_{10}	1	`float`	Average over all inputs bit width
Output slew	y_1	4	`float`	Transition times for output pin
Delay	y_2	4	`float`	Delay from current to output pin

[1] If the training set is DS-I, then $T = 20$. Otherwise, $T = 37$.
[2] If the training set is DS-I, then $B = 12$. Otherwise, $B = 32$.
[3] Extra features considered for DS-II.

Moreover, the output pin's slew is also determined by an LUT, indexed by the fan-out load and input slew. This slew is then passed through the timing graph so that the output slew of one component becomes the input to the next connected pin. Drawing inspiration, this thesis evaluates the model's performance with and without taking slew into account, treating it as a feature and the output slew as an objective. Thus, a sequential inference ensures proper consideration of the slew propagation through the timing graph.

The datasets having tabular and graph structures require similar but differently defined features. In the following, features for both dataset types are introduced.

6.3.1. Tabular Features

The selected features x and objectives y are listed in Table 6.2. The nature of the STA task inspires the choice of these features and objectives. Especially the features are picked for the following reasons:

1. Component type (x_0): The component type at the RTL IR gives an intuition about the component's complexity. It also correlates to implementation patterns in the transistor netlist.
2. Input number (x_1): It identifies the pin-to-pin connection from the current pin to the output among all input pins considered as x_2. It highlights the signal's path through the component and design. It is relevant, especially if inputs have a different function, like the multiplexers.
3. Number of inputs (x_2), input bit width (x_3), and output bit width (x_4): The number of inputs x_2 and their sizes or bit widths x_3, x_4 influence the time needed to execute a function. The number of inputs also correlates to the implementation pattern.
4. Number of connections to input (x_5) and output (x_6) pins: NLDMs define the gate delay as a function of the gate's capacitive load. This is proportional to the number of

Table 6.3.: Example of tabular features for the sample MoD in Figure 5.8a.

Component	**Input**	**Output**	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_8	x_9	x_{10}
BAND0	In00	Out	BAND	0	3	1	1	1	3	1	3	3
	In01	Out	BAND	1	3	1	1	1	3	1	3	3
	In02	Out	BAND	2	3	1	1	1	3	1	3	3
BAND1	In00	Out	BAND	0	2	1	1	1	3	1	2	2
	In01	Out	BAND	1	2	1	1	1	1	1	2	2
BXNOR0	In00	Out	BXOR	0	2	1	1	1	3	1	2	2
	In01	Out	BXOR	1	2	1	1	1	1	1	2	2

connections to the pin.

5. Slew at the input pin (x_7): The gate's output slew depends on the slew model, a function of the input slew and the fan-out. The slews for primary ports are defined as timing constraints. The slew for other connections is propagated.
6. Number of outputs (x_8): The number of outputs correlates with the component implementation. It should be considered for multiple output components like the demultiplexers.
7. Bit widths of all input pins added (x_9) and averaged (x_{10}): The bit width of other input pins, different from the one indicated with x_1, gives information about the complexity of the logic mapping.

For both tabular datasets, the categorical feature x_0 is encoded using a one-hot format and embedding layers as depicted in Figure 3.2. Following the one-hot encoding, x_0 is mapped to an $N \times T$ sparse matrix, where N is the number of samples and T is the number of component types supported, i.e., 20 for DS-I and 37 for DS-II. Additionally, this work maps the bit widths per input to a zero-padded vector, resulting in an $N \times B$ matrix, where $B = 12$ and $B = 32$ for DS-I and DS-II, respectively. A *MinMaxScaler* normalizes the features and four objectives in the $[0, 1]$ range.

For completeness, given the MoD in Figure 5.8a, Tables 6.3 and 6.4 show the proper values for the selected features and ground-truth labels, respectively. In particular, the values in bold in Table 6.4 are directly extracted from the timing report in Figure 5.8b. This MoD consists of a main BAND gate with three input pins of one bit and a fan-out equal to three. Figure 5.8a also highlights the pin-to-pin features extracted and represented in a tabular structure, where each row is a pin-to-pin connection on the MoD, and each column represents one of the features explained in Table 6.2. The input slews for the primary inputs are known and configured in the input SDC file. As these values are not predicted, they are not directly shown in this table.

Moreover, the analysis methods discussed in Section 3.1.2 are applied to verify the correct selection of features and objectives. Precisely, the SRCC is calculated for both datasets. Moreover, the MI and Shapley Values for DS-II are calculated, specifically for the features x and the two objectives, output slew y_1 and delay y_2 distinguished into rising (r) and falling edge (f). For simplicity, only the results for the falling edge, i.e., y_{1f} and y_{2f} are shown. The results for the rising edge highlight similar trends described below. The only difference is that the dependence between the input slew feature for the rising edge x_{7r} and objectives with the same

Table 6.4.: Example of tabular ground-truth labels for the sample MoD in Figure 5.8a.

Component	Input	$y_{1,0}$	$y_{1,1}$	$y_{1,2}$	$y_{1,3}$	$y_{2,0}$	$y_{2,1}$	$y_{2,2}$	$y_{2,3}$
BAND0	In00	0.0	0.0693	0.0416	0.0374	0.0	0.13	0.1674	0.1324
	In01	0.0713	0.0693	0.0416	0.0374	0.1431	0.127	0.1472	0.1181
	In02	0.0	0.0693	**0.0416**	0.0374	0.0	0.1307	**0.1771**	0.1396
BAND1	In00	0.0196	0.0159	0.0165	0.0127	0.0647	0.0643	0.0723	0.0707
	In01	0.0196	0.0159	0.0165	0.0127	0.0788	0.0681	0.1131	0.0868
BXNOR0	In00	0.0674	0.0184	**0.0371**	0.0164	0.0593	0.0578	**0.0847**	0.0334
	In01	0.0674	0.0184	0.0371	0.0164	0.1168	0.0784	0.1372	0.0485

edge type, i.e., y_{1r} and y_{2r}, is stronger than with the falling edge x_{7f}. In the following feature analysis, the categorical feature x_0 was not included.

6.3.1.1. SRCC

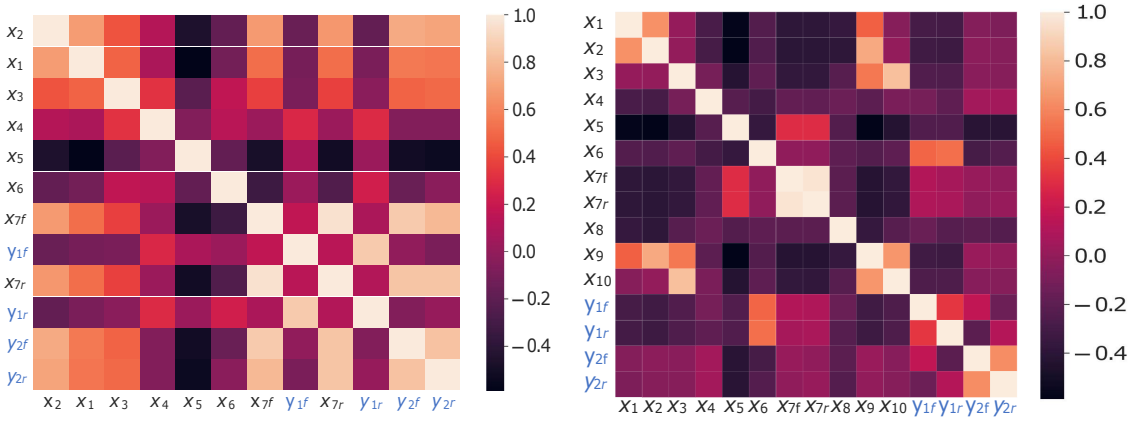
To analyze the correlation between features and objectives from Table 6.2, the SRCC in Equation 3.1 is applied over all pairs. Figure 6.4 shows the SRCC matrix calculated for both datasets. The lighter the color in the heatmap, the higher the correlation between a pair of variables is. Objectives are highlighted in blue for both axes.

For DS-I, most features x correlate with the objectives y. The lower correlation between x_5 and other features and objectives is because only one component type, the DEMUX, of the dataset has multiple outputs. The number of outputs for the other component types is fixed to one. Thus, feature x_5 does not really impact the delays and slews across the different component types. As expected, there is a high correlation between y_2 and x_7 features and structure-related features such as x_1, x_2, and x_3.

Similarly, the features x correlate with the objectives y for DS-II. As DS-II is larger than DS-I and contains more variety of component types, the correlation among pairs of variables is reduced. This is because the correlation for certain variables is lower in certain component types than in others. So, the average correlation factor gets reduced. For instance, the correlation between the number of input pins and the selection bit width in a MUX differs from the DEMUX.

6.3.1.2. Mutual information (MI)

The MI method analyzes the linear and non-linear relationship between the selected variables and the objectives to predict. The MI in Equation 3.2 is computed for each selected feature in Table 6.2 and the regression objectives: the output slew and the delay. Figure 6.5 presents the MI statistic results depicting the relationships between the features x and the output slew y_1, as well as the delay y_2 for the falling edge. Through this analysis, the MI highlights the input bit widths x_9 and x_{10} as the features with the strongest correlation to the objectives. Furthermore, it is revealed that the objectives and the number of outputs x_8 exhibit a less significant mutual dependence.



(a) DS-I (b) DS-II

Figure 6.4.: Spearman's correlation matrix for both tabular datasets.

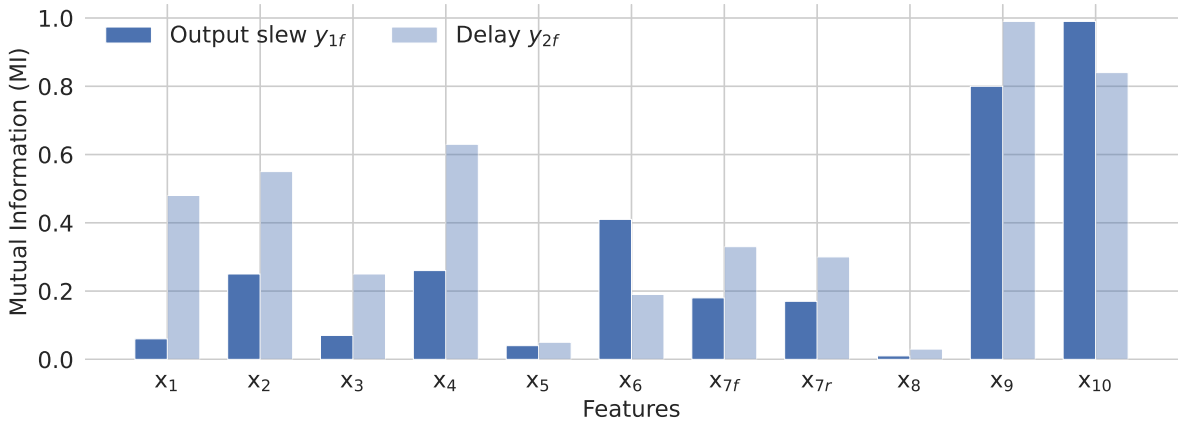
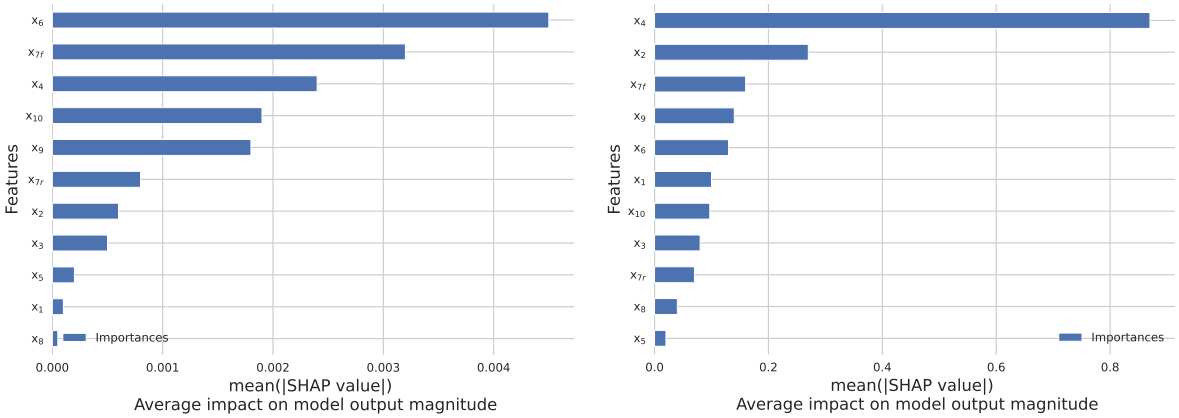


Figure 6.5.: Mutual information values for features and objectives for the falling edge in DS-II.



(a) Output slew y_{1f} (b) Delay y_{2f}

Figure 6.6.: Shapley values for the falling edge values in DS-II.

6.3.1.3. Shapley Values

The Shapley values highlight the impact or importance of each selected feature in the prediction of each objective. To that end, an XGBoost [147] regressor model is used as an explainer. Figure 6.6a shows that the most expressive features for the output slew prediction are related to the features fan-out x_6 and the input slew x_{7f}. The latter is in sound with the definitions of NLDMs used by STA tools. For the delay prediction, RTL-related features, such as the bit width of the output port x_4 and the number of inputs x_2, have higher Shapley values, as shown in Figure 6.6b. The input slew is again among the top five most important features.

The results obtained from the statistical methods shown in Figures 6.4, 6.5, and 6.6 confirm the intuition that there exist both linear and non-linear relationships and high correlation between high-level features from RTL IRs and timing metrics, such as pin-to-pin delays and slews. These results validate the notion that features extracted directly from RTL IRs can be highly informative in predicting the timing behavior of their corresponding gate-level representation.

6.3.2. Graph Features

Once the graph topology has been determined, numerical features are assigned to both nodes and edges, aligning with the mathematical definition of featured graphs as outlined in Section 3.1.4.3.1. These graph features resemble those found in tabular datasets and encompass attributes such as component type, the count of input and output pins, and bit widths. Notably, a specific feature indicating the current pin is considered redundant when the entire graph is utilized as a training sample. Additionally, the depth level of a node in the graph is incorporated as a feature to accommodate both simple and complex graphs. The selection of graph features is based on their strong correlation with features in tabular datasets, thereby yielding comparable outcomes when employing feature analysis methods.

Table 6.5 shows the node features followed by the input slew as the only edge feature. The

Table 6.5.: Features and objectives for nodes and edges of graph dataset DS-III.

Feature & Objectives	Symbol	Size	Type	Description
RTL node type	x_0	23	`int`	One-hot encoded node type
No. Inputs	x_1	1	`int`	Number of input connections to the node
No. Outputs	x_2	1	`int`	Number of output connections of the node
Bit width	x_3	1	`int`	Bit width of the node
Depth level	x_4	1	`int`	Level of the node in the DAG
Edge inverter	x_5	1	`int`	Whether the component inverts the timing arcs
Input slew	x_6	4	`float`	Input transition time from the input pin of the edge
Output slew	y_1	4	`float`	Output transition time of the output pin of the edge
Delay	y_2	4	`float`	Delay through the gate from input to output pin

Listing 6 Example of generated GML file in Figure 5.8a.

```
# start graph
graph
[
  hierarchy 1
  directed 1

  # start list of nodes
  node  [
    id 0
    label "Top"
    x_0 0
    x_1 0
    x_2 5
    x_3 0
    x_4 0
  ]
  node  [
    id 1
    label "End"
    x_0 0
    x_1 2
    x_2 0
    x_3 0
    x_4 5
    x_5 0
  ]
  node  [
    id 2
    label "Top/In00"
    x_0 1
    x_1 1
    x_2 1
    x_3 1
    x_4 1
    x_5 0
  ]
  ...
  node  [
    id 19
    label "Top/BAND1/Outp"
    x_0 3
    x_1 2
    x_2 1
    x_3 1
    x_4 4
    x_5 0
  ]

  # start list of edges
  edge
  [
    source 0
    target 2
    x_6 [0.0, 0.0,
         0.0, 0.0 ]
    y_1 [0.5, 0.2,
         0.5, 0.2]
    y_2 [0.2, 0.1,
         0.2, 0.1]
  ]
  edge
  [
    source 16
    target 17
    x_6 ...
    y_1 ...
    y_2 ...
  ]
  ...
]
```

edge labels are the output slew and delay and become the objectives annotated to all graph's edges. Each edge label considers rising and falling edges and minimum and maximum corners. To ensure uniformity, both node and edge features and labels are normalized to fit within the $[0, 1]$ range. The node feature indicating the component type is one-hot encoded, resulting in a vector of size 1×23. Notice that the number of RTL-supported components is 19. The remaining four types encompass primary port pins, start-end nodes, or unknown components, which might be encountered while evaluating unseen designs. Hence, the node features are one-dimensional vectors comprising 28 elements, while the edge feature vectors have dimensions of 1×4. Similarly, the edge label vectors are sized at 1×8, ensuring a comprehensive representation of the graph's attributes and objectives.

For completeness, given the MoD in Figure 5.8a, Listing 6 shows a snippet of the created graph saved in a GML file. Nodes and edges are annotated with proper values for the selected features and labels.

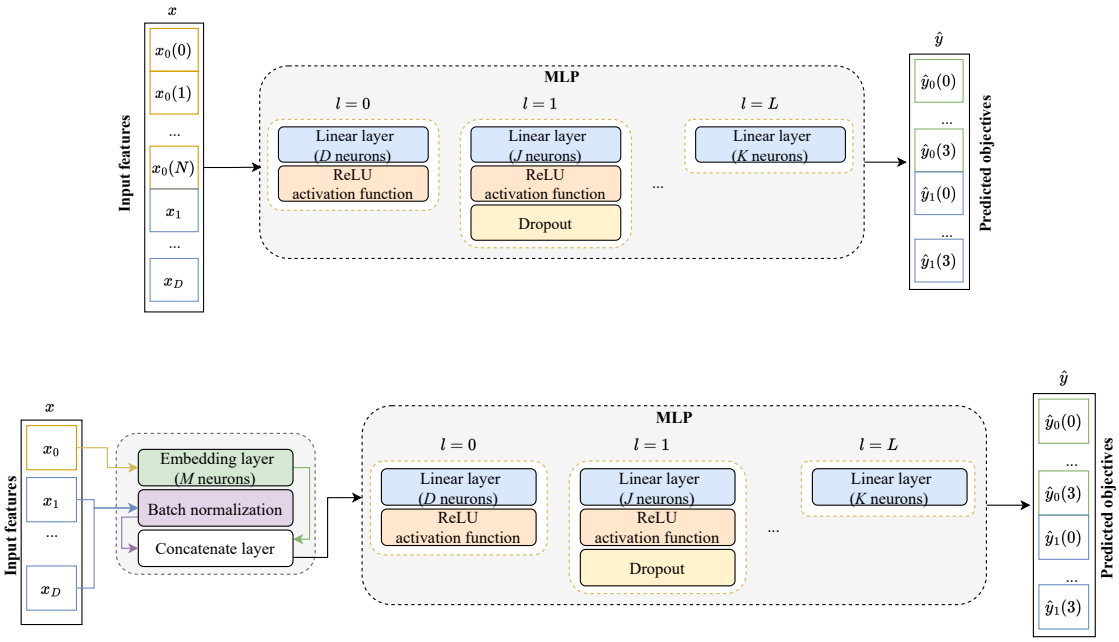


Figure 6.7.: Implemented MLP-based models for the pin-to-pin delay estimation.

6.4. Machine Learning Models

The next step in the ML flow is selecting and training ML models. This thesis proposes both vector- and graph-based ML models to cope with the data structure of the generated datasets.

Tabular datasets are commonly used in ML applications. For this type of dataset, MLP networks are employed. An MLP processes the input features of each row in the dataset with multiple linear layers. Even a simple MLP can represent a broad range of functions with appropriate hyperparameters. On the other hand, when dealing with graph-based datasets, initial processing is done using GNNs [71]. As introduced in Section 3.1.4.3, GNNs are neural networks that condense the graph's topological and feature information into lower or higher dimensional vectors. The resulting encoded vectors are inputs to an MLP, which is then trained for a regression task at the edge level. Subsequently, additional information regarding the selected models is provided.

6.4.1. MLP-based Models

For the tabular datasets built as in Figure 6.2, this thesis compares and implements two main architectures: a simple *MLP* [73] and the tabular *TabMLP* [148]. Both are portrayed in Figure 6.7 and explained in detail in the following sections.

6.4.1.1. A Simple MLP

The architecture of the simple MLP is illustrated in Figure 6.7a. An MLP comprises essential layers, including the input, hidden, and output. The number of neurons in the input layer, D, equals the number of input features, while the output layer has a number of neurons, K, equal to the number of objectives. The number of hidden layers $L-2$ and their corresponding number of neurons J are hyperparameters to be fine-tuned. All the layers are fully connected, meaning each element in the subsequent layer results from applying the activation function to the weighted sum of all values in the current layer. Each hidden layer uses a Rectified Linear Unit (ReLU) [149] activation function. Lastly, the two first hidden layers employ a dropout layer to avoid overfitting.

Input features must be numerical values. However, the component type feature x_0, indicating the gate's functionality, is a categorical variable or string. As already discussed, one-hot encoding is commonly used to generate sparse binary vectors of size N, where N is the total number of categories. The MLP in Figure 6.7a uses this encoding format for x_0, transforming it into a vector of size N. Additionally, the bit width for all inputs is zero-padded, resulting in a $1 \times B$ vector. To compute the total number of input features D, the values N, B, and the remaining features listed in Table 6.2 are accumulated.

Training the MLP model with DS-I and DS-II, with and without considering slew, results in three models with different configurations. These different MLP-based models and their respective configurations are documented in Table 6.6. For DS-I, the number of input neurons D becomes 41 and 43 when input slew features are considered or not, respectively. Correspondingly, the number of outputs K becomes four and eight, considering slew and delay for the four possible timing edges. However, the MLP model trained with DS-II always considers input slew features, resulting in the number of output neurons K being fixed at 8. As the supported component types and the maximum number of inputs increased, the input layer of this MLP model has $D = 80$ neurons.

Finally, using the TPE approach and a validation dataset, as explained in Section 3.1.5, the best hyperparameter values J and L are found. Moreover, the hyperparameter search is also employed to find the best learning rate α and the dropout rates per model.

6.4.1.2. Tabular-MLP

The TabMLP architecture depicted in Figure 6.7b uses an embedding layer encoding the categorical feature x_0 into an N-dimensional vector. As discussed in Section 3.1.3, embedding layers get the numerical value per each category as inputs and, during training, learn the best N-dimensional encoding vector for each sample.

In Figure 6.7b, the resulting embedding vector of x_0 is concatenated with the rest of the features passed through a batch normalization layer. Afterward, the concatenated vector is passed through an MLP model. For DS-I, the value of N equals 10, with the total number of numerical input features amounting to 21 and 23, with and without considering the slew,

Table 6.6.: Configuration details of tabular models and best-found hyperparameters.

Model	Dataset	Slew	No. Neurons		Hyperparameters			
			Layer 0 D	Layer L K	No. Hidden layers $L-2$	No. Neurons[1] J	Dropout ratio [2]	α
MLP	DS-I	✗	41	4	3	[39, 17, 62]	[0.04, 0.01]	0.008
		✓	43	8	2	[111, 59]	[0.07, 0.08]	0.040
	DS-II	✓	80	8	2	[256, 255]	[0.037, 0.27]	0.001
TabMLP	DS-I	✗	21	4	3	[210, 145, 84]	[0.12, 0.11]	0.027
		✓	23	8	3	[144, 102, 74]	[0.17, 0.24]	0.029
	DS-II	✓	43	8	3	[256, 225, 128]	[0.22, 0.32]	0.005

[1] Number of neurons in each hidden layer.
[2] Dropout rate for the first two hidden layers.

respectively. The number of neurons in the output layer is four. In the case of DS-II, the value of N is 19, aligning with the expanded supported ranges and the consideration of slew features. This results in a total number of numerical input features of 43 and four neurons in the output layer. The configuration of these three implemented MLP-based models, along with the best hyperparameters found, is summarized in Table 6.6.

6.4.1.3. Training Results

In total, six models are built as listed in Table 6.6. These models result from combining both structures, MLP and TabMLP, both datasets, DS-I and DS-II, and with and without considering slew. When training with the dataset DS-II, the slew feature is always considered. The latter is motivated by the feature analysis results in Section 6.3 and the task's nature detailed in Chapter 3. Slew is essential as it is estimated and propagated across all timing paths.

Training is carried out after the model architectures are defined and the hyperparameters are fine-tuned. The models are trained in a NVIDIA® Tesla® P40 GPU for 300 epochs. For the MLP model, the overall training time when using DS-I is 164.9 seconds, and for the TabMLP model, 706.5 seconds. For DS-II, the training times are 3.7 and 4.4 hours for the MLP and TabMLP models, respectively. Figure 6.8 shows the results when evaluating each model over the test dataset. The test dataset consists of generated designs using one gate with zero and up to six connected gates, like the training dataset. The models achieve an R^2 coefficient higher than 86%, even without considering slew features or propagation. However, actual designs are much more complex. Section 9.1 presents evaluation results using the six pre-trained models over more realistic hardware designs.

6.4.2. GNN-based Models

The inputs to the MLP and TabMLP models are feature vectors per each pin-to-pin connection in a design, as shown in Figure 6.2. This mapping of EDA objects to Euclidean data (e.g., vectors or matrixes) has been done to cope with state-of-the-art ML techniques, such as NNs

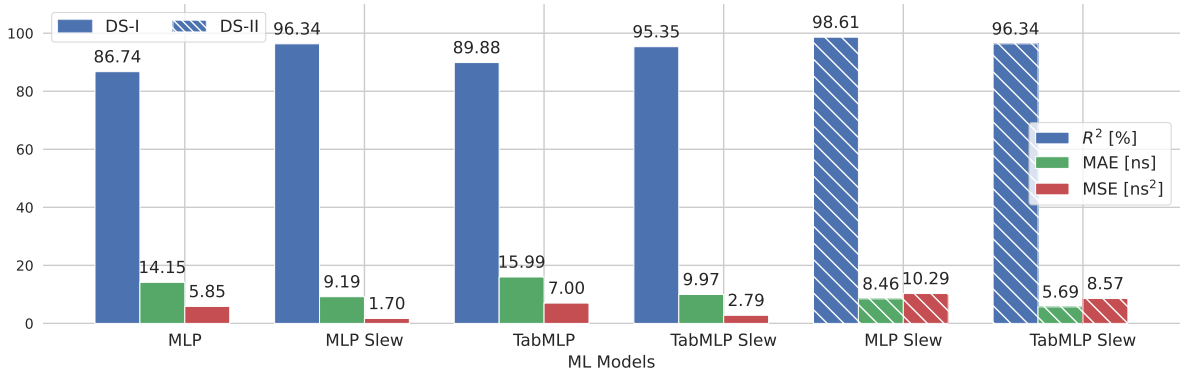


Figure 6.8.: Average estimation performance of the different models over the test dataset.

or CNNs. However, it does not consider that EDA objects such as circuits or netlists are intuitively featured graphs. In response, GNNs are introduced as an NN framework that operates directly on graphs.

Section 3.1.4.3 presents a detailed definition of graphs and GNNs. At a high level, given an input featured graph, GNNs learn to combine feature and topology information and provide encoded embedding representations per nodes, edges, or graph. For instance, the node embedding vector is denoted as $\mathbf{h}_u$ $\forall u \in \mathcal{V}$ and encodes the neighborhood information of each node u [85]. GNNs can handle unseen nodes and consider both topological and feature information. They employ a message-passing strategy, where node embeddings or messages are transmitted through neighboring nodes, and a non-linear differentiable function is applied to update the node embeddings. Consequently, the feature information is treated as a message and is propagated through the local neighborhood, guided by the topological information.

For GNNs, the tabular datasets from Figure 6.2 are transformed to an intuitive graph dataset following the structure in Figure 6.3. Each graph is represented by its topology using a list of edges or the adjacency matrix $\mathbf{A}$. Moreover, all feature nodes x_n $\forall n \in \mathcal{V}$ are stacked to build a matrix $\mathbf{X}$. Similarly, the edges are annotated with features x_e and stacked into a matrix $\mathbf{X^e}$. Ground-truth labels are annotated to the edges and represented as y_e vectors.

To process this graph-based dataset, the extended GCN [150] and an MPNN [151] architectures in Figure 6.9 are used. Both architectures are chosen as they obtain the node embeddings while considering multidimensional node and edge features. As shown in Figure 4.3, GNN-based models require a GNN to extract the embeddings of the graph and an MLP predictor to accomplish an end-to-end task, which can be classification or regression at node, edge, or graph level. The following explains the GNN architectures and the MLP predictor used.

6.4.2.1. Graph Convolutional Network (GCN)

Graph Convolution Networks (GCNs) are the most commonly used architectures. They belong to the category of ConvGNNs, which are a generalization of convolution applied to graphs, as

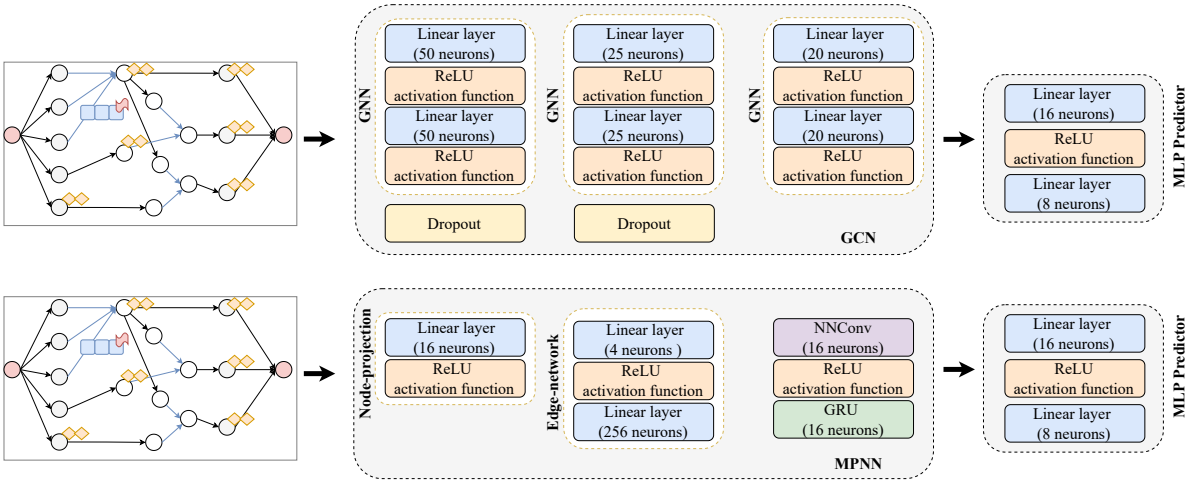


Figure 6.9.: Implemented GNN-based models for the pin-to-pin delay estimation.

explained in Section 3.1.4.3.3. GCNs are considered spectral operators, which implement the convolution of two signals in the time domain as the multiplication of these two signals in the frequency domain. The proposed GCN-based model is a set of stacked GCN layers L. Mathematically, the propagation along L layers of GCN type follows Equation 6.1:

$$\mathbf{H}^{(l+1)} = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right), \tag{6.1}$$

where $\mathbf{H}^{(l)}$ is the stacked matrix of vector embeddings h per n nodes in layer l. In $l = 0$, the embeddings of the nodes are their feature vectors, and $\mathbf{H} = \mathbf{X}$. Thus, $\mathbf{H} \in \mathbb{R}^{n \times d}$. $\sigma(\cdot)$ is a non-linear activation function. The formulation of GCNs adds self-connections to each node of the initial graph by adding the identity matrix of order I_n to $\mathbf{A}$, i.e., $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I_n}$. The new $\tilde{\mathbf{A}}$ is multiplied side-by-side with its diagonal degree matrix $\tilde{\mathbf{D}}$. The $\mathbf{W}^{(l)}$ matrix contains the learnable parameters of the l layer.

The original definition of GCN has the drawback of not supporting message passing with multidimensional edge features, i.e., the edge feature matrix $\mathbf{X_e}$ is not considered in Equation 6.1. To overcome this, an extension of GCNs is used to update node embeddings with multidimensional edge feature vectors. The used GCN architecture, depicted in Figure 6.9a, includes three GNN and two dropout layers. Each GNN layer is composed of two linear layers. Mathematically, the first layer creates a message $\mathbf{m}$ by concatenating node and edge features, as in Equation 6.2.

$$\mathbf{m}_u = \sigma(\mathbf{W}(\mathbf{h}_u || \mathbf{he}_{(u,v)} || \mathbf{h}_v) + b), \tag{6.2}$$

where, $\sigma(\cdot)$ is the activation function, W and b are the trainable parameters of the layer, and $||$ is the concatenation operator. Notably, the linear layer concatenates the embeddings of the edge connecting a source u and target v node, as well as their respective node embeddings h_u and h_v. The second linear layer updates the current node embedding, concatenating the previous embedding vector and the current built message as described in Equation 6.3.

$$\mathbf{h}_u = \sigma(\mathbf{W}(\mathbf{m}_u || \mathbf{h}_u) + b) \tag{6.3}$$

Figure 6.9a depicts the use of the ReLU activation function following each linear layer while dropout is applied to the first GNN layers. The number of neurons per layer is also displayed in this figure. In particular, 50 input neurons are utilized, accounting for the combined number of node features, which are counted double for h_u, h_v, and the edge features in $\mathbf{he}_{(u,v)}$.

6.4.2.2. Message Passing Neural Network (MPNN)

Message Passing Neural Networks (MPNNs) are also part of the ConvGNNs. In contrast to GCNs, MPNNs support per definition edge features $\mathbf{x_e}$. To that end, a MPNN builds the message m_u that travels to the node u from its neighbors $v \in N(u)$ and considers the edge features $x^e_{(u,v)}$, as shown in Equation 6.4:

$$m_u = \sum_{v \in N(u)} M(h_u, h_v, x^e_{u,v}), \tag{6.4}$$

where $M(\cdot)$ is the message function, usually implemented as an MLP. All messages arriving at the node u are aggregated using the addition operator. The message m is then used to calculate the updated node embedding h_u following Equation 6.5.

$$h^{l+1}_u = F(h^l_u, m^{l+1}_u), \tag{6.5}$$

where $F(\cdot)$ is a node update function also usually implemented as a simple MLP.

The building blocks of the implemented MPNN architecture are shown in Figure 6.9b. First, the node features are passed through a linear layer and a ReLU function for an initial *node projection*. Afterward, the message-passing strategy is performed two times. Each time, an *edge network* composed of two linear layers and a ReLU function transforms the edge features and their hidden embedding vectors. The resulting edge embeddings are the inputs to an *NNConv*, a graph convolution layer that multiplies its inputs with the node embeddings of the neighbors. This product is added to the embeddings of the current node. Finally, a Gated Recurrent Unit (GRU) [152] is used to recurrently combine the current node embeddings with the node features [153].

6.4.2.3. MLP Predictor

The node embeddings obtained by the modified GCN and the MPNN are the inputs to an MLP predictor, as shown in Figures 6.9a and 6.9b, respectively. Specifically, the input to the MLP predictor is a concatenated vector of the embedding vectors of the source nodes h_{src} and destination nodes h_{dst} of the edges. Thus, the input dimension of the MLP is $1 \times 2D$, where D is the size of the output node embedding vector. The number of neurons in the output layer is eight, so the MLP predicts the eight edge labels listed in Table 6.5, i.e., the maximum and minimum output slews and delays for rising and falling edges.

The end-to-end frameworks using a GNN and an MLP predictor are trained in a supervised setup with a stochastic gradient descent optimizer. Barboza et al. [130] state that the loss

function for slew and delay prediction should penalize underestimations, i.e., the cases when the predicted $\hat{y}$ value is lower than the truth y value. Thus, they introduce the *asymmetric MSE* loss function formulated in Equation 6.6. The first term in this equation corresponds to the conventional MSE loss defined in Equation 3.6, and N is the total number of samples. Finally, both models are trained on an NVIDIA® Tesla® P4 GPU for 100 epochs.

$$\text{Asymmetric MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \frac{1}{N}\sum_{i=1}^{N}(|y_i - \hat{y}_i| + y_i - \hat{y}_i)^2. \tag{6.6}$$

6.5. Summary

This chapter proposes a novel methodology for early pin-to-pin estimations at the early stages of the ASIC design flow. In essence, the chapter estimates delays and slews of pin-to-pin connections inside an RTL IR using ML. This involves leveraging the training circuits and timing reports collected using the automated flow in Chapter 5 to construct tabular and vector-based datasets. Subsequently, the ML methods explained in Chapter 3 are employed to address the estimation task. In particular, feature analysis and engineering are performed to validate and transform the selected features into a numerical format. ML models are implemented to cope with the dataset structure, evaluating different architecture flavors and feature sets. The employed models vary from classical networks, such as the MLPs, to modern GNNs. Notably, models with and without considering the slews of the input signals are evaluated, with the latter aiming to closely simulate the actual process conducted by STA tools.

In contrast to the work reviewed in Chapter 4, this methodology is the first to input RTL IRs and estimate post-logic synthesis values. This methodology leverages state-of-the-art ML flows and carefully selects features and data structures to match the actual nature of the timing analysis flow. Pin-related features, fan-outs, and slews are evaluated as features in this sense. Additionally, the pin-to-pin connections of an RTL IRs and their features are represented in a tabular way. Further, the complete RTL IRs are mapped to intuitive graph representations.

The methodology outlined in this chapter offers significant contributions. Firstly, it exploits the interconnections between RTL generation frameworks, open-source synthesis tools, and ML methods. RTL generation frameworks generate training circuits from which features and HDL objects can be obtained. The latter are synthesized, and the generated reports are used as sources of ground-truth labels. Features and labels are fed to an ML-based flow to do early, fast, yet accurate design metric predictions. Even though such a flow sounds intuitive and logical, this thesis is the first work in this direction. The initial idea of such a flow was already considered in 2018 when Hashemi et al. proposed METRICS2.0 [140] as an updated infrastructure for circuit data collection and sharing for ML applications. However, just in 2021, METRICS2.1 and Chisel were connected to the OpenROAD toolchain and integrated into the Robust Design Flow (RDF-2021) [142]. This validates the proposed flow in this thesis. Nevertheless, to the best of our knowledge, ML applications using the RDF-2021 flow solely employ the generated HDL files, neglecting the advantage of the IR provided in Chisel.

Moreover, thanks to the use of an RTL IR representing a design as a set of primitive sub-

components, the proposed models are trained to predict the timing behavior of those sub-components. These are the building blocks of more complex and unique designs. In contrast to the related work in Chapter 4, the generated training circuits are not actual designs but instead small circuits with different primitive components and configurations. Thus, the models do not overfit the designs used in training. Additionally, the training and evaluation circuits are not limited, as far as they are composed of the component types supported during training.

The data and models in this chapter are built under fixed constraints regarding the input circuits and STA configurations. However, the same flow can be applied when using different RTL generation and synthesis conditions. In particular, targeting different technology nodes, operating conditions, tool parameters, timing constraints, or different target design metrics mostly require a re-iteration of the complete flow from data collection to the re-training of the models. Thus, this chapter's methodology becomes essential for an automated flow for data generation, feature analysis, preprocessing, hyperparameter search, and model training.

Lastly, the pre-trained models find the delay and slew values for each pin-to-pin connection in a design. Higher delay values imply critical pins where the signal is too late. As the inputs to the models come from the RTL IRs in a tabular or a graph format, the HDL generation step is skipped. In other words, critical pins can be detected without HDL generation, synthesis, and STA runs.

While utilizing ML models for inference offers expedited results compared to the traditional flow involving EDA tools, the process of training, validating, and evaluating a model for a specific task presents significant challenges. The models and experiments discussed in this chapter show the complexities inherent in the task of delay estimation, mainly when applied to more complex circuits, as outlined in Section 9.1. Hence, this chapter's primary contribution lies in framing the timing estimation as a regression task, facilitating the derivation of crucial conclusions. Firstly, it emphasizes the necessity for more strict configurations for training circuits to cover the required configurations rather than all possible ones. Secondly, it recognizes the diverse nature of actual designs, which employ varied component types with distinct features and timing behaviors, thereby advocating for the training of dedicated ML models for each component type. Finally, it highlights the importance of considering different timing constraints regarding the delay and input slews of primary ports when generating the training datasets. These conclusions serve as fundamental considerations for the subsequent chapters of this thesis.

7. Maximum Arrival Time Estimation

Early knowledge of timing metrics allows pinpointing critical components of the design and, thus, optimization in the design flow. Chapter 6 describes the ML flow when training models for the delay and slew estimation of pin-to-pin connections. However, these estimations do not provide sufficient information about critical components or paths in the complete design. State-of-the-art works reviewed in Chapter 4 use ML to predict global design metrics such as TNS [134] or ATs [133]. However, there are two different approaches. They train ML models to estimate global timing metrics directly, or they leverage the pin delay and slew estimation to estimate or calculate global design metrics as in [135], [136]. The latter approach is inspired by the natural STA flow and is followed in this chapter, which focuses on estimating the maximum AT of a design and leverages the delay and slew estimation from Chapter 6.

Specifically, this chapter proposes GNNs to predict the AT of RTL IRs after logic synthesis. Using the data generation flow described in Chapter 5, training and evaluation designs are generated. From the RTL IRs, graphs are built and annotated with features per nodes, edges, and graphs. GNNs embed feature and topological information of graphs and are trained to estimate not only the edge information as in Chapter 6, but also the graph's worst AT overall design endpoints. This work is the first to integrate component delay and slew with AT prediction as in [135] and [136] but at the higher abstraction level of RTL. The following sections define the estimation problem and the key steps to solve it using the ML methods from Chapter 3 and the main findings from Chapter 6.

7.1. Problem Statement

Given an RTL IR comprising combinatorial logic components, estimations of global timing metrics such as the maximum AT are required to pinpoint its critical regions at the early stages of the design flow. For instance, in the schematic view of Figure 7.1, delay and slew values must be predicted for all the pin-to-pin connections highlighted in blue. Furthermore, the maximum AT overall design's endpoints can give information about the critical regions of a design.

Similar to Chapter 6, this chapter uses GNNs to encode featured graphs. However, the resulting encoding vectors are used to solve two regression tasks simultaneously: the delay and slew prediction of single components, and the prediction of the maximum AT. At a high level, the building blocks of the flow in this chapter are depicted in Figure 7.2. RTL IRs are mapped to DAGs and annotated with multidimensional edge, node, and graph features. Ground-truth values for ATs, slews, and delays are extracted from timing reports generated by open-source tools. Graphs are encoded using GNNs. The following sections provide details of the implementation of those steps applied to both regression tasks.

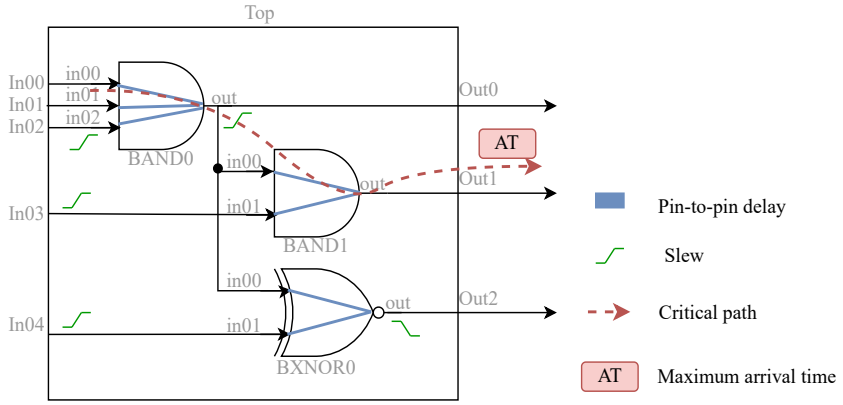


Figure 7.1.: RTL IR of a combinatorial circuit with pin-to-pin delays, slews, and maximum arrival times.

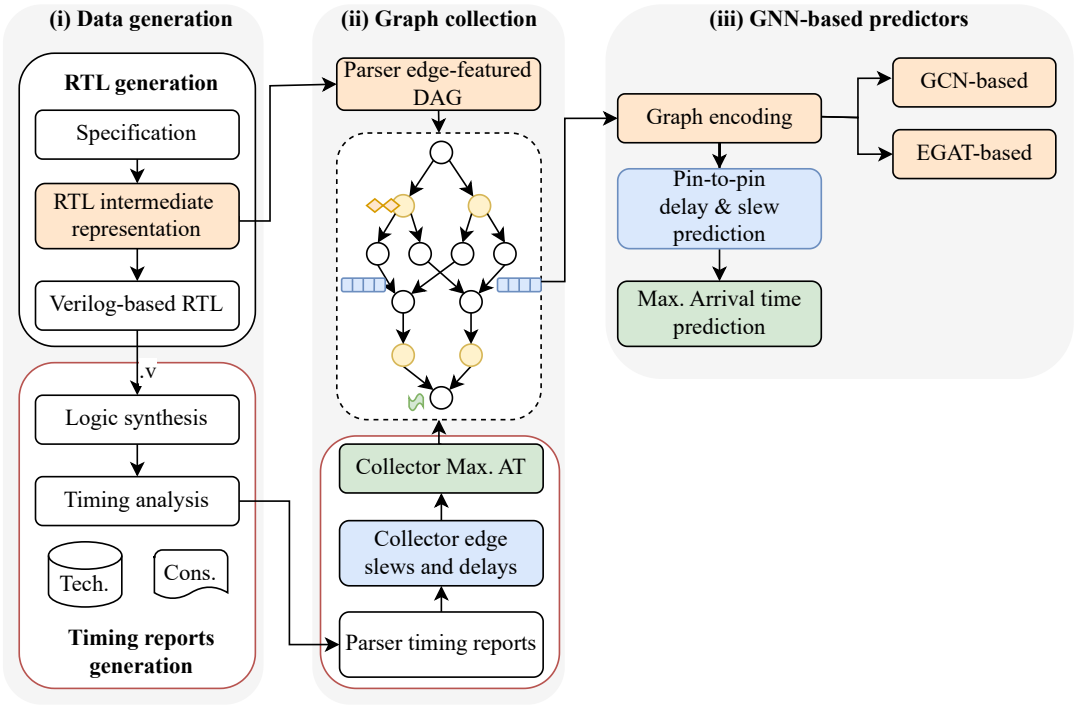


Figure 7.2.: Building blocks of the flow predicting delays, slews, and ATs from an RTL IR.

7.2. Data Generation

The first step in the flow depicted in Figure 7.2 is data generation. This section uses the data generation flow described in Chapter 5 but with minor adaptations regarding the initial specifications, the number of OpenSTA runs, and the extracted ground-truth labels. These adaptations are based on the findings of Chapter 6 and aim to increase the quality of the models to perform better when tackling a more challenging task such as the AT estimation.

First, the random specifications comprise at least one of the nineteen supported logic gates, such as NOT, AND, OR, and XOR, in reduced, logical, and bitwise fashion. A total of 4.9K designs are generated, encompassing common configurations of logic gates used in existing hardware modules. Table A.2 lists more details about the selected configurations for logic operators.

The generated RTL designs are synthesized using a proprietary 40 nm technology. Synthesis is conducted hierarchically, ensuring the boundaries of the RTL modules are preserved. Using the synthesis flow in Figure 5.6, OpenSTA is executed 25 times concurrently for each netlist. Each run targets different delays and slews for the primary ports. In total, 25 random maximum and minimum delays are used in the ranges $[0, 2]$ and $[0, 1]$ and for input slews $[0, 1]$. These different values facilitate capturing a wide range of slews, delays, and ATs.

Using OpenSTA, reports are generated for timing paths passing through each component's input and output pins in the RTL IR. Generated timing reports are collected and parsed. For each pin-to-pin connection in the RTL IR, delay and slew values are extracted. In contrast to Chapter 6, the ground-truth labels correspond not only to the pin-to-pin delay and slew information but also the maximum AT over all the endpoints in a design.

7.3. Graph Collection

After generating the RTL IRs and their corresponding timing reports, the subsequent step is to build and collect featured graphs that portray the nature of the regression task. As shown in Figure 7.2, the RTL IRs are transformed into a featured DAG. This representation closely resembles the graph structure described in Section 6.2.2. However, this chapter extends the definition of featured graphs outlined in Section 3.1.4.3.1 to consider vectors of graph features. In this context, a ***featured graph*** is defined by a tuple $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \mathbf{X}, \mathbf{X_e}, \mathbf{g}\}$, where in addition to the set of nodes $\mathcal{V}$, edges $\mathcal{E}$, and matrices for node and edge features $\mathbf{X}$ and $\mathbf{X_e}$, a new vector $\mathbf{g}$ is added to represent graph-level features. Furthermore, edges and graphs are annotated with ground-truth labels $\mathbf{y}^{\mathcal{E}}$ and $\mathbf{y}^{\mathcal{G}}$, respectively. GNN-based models are trained to predict $\mathbf{y}^{\mathcal{E}}$ and $\mathbf{y}^{\mathcal{G}}$ for unseen graphs through supervised learning, utilizing a training set of graphs.

Inspired by the features employed for the dataset DS-III, the node and edge features in this chapter are derived from the attributes outlined in Table 6.5. Additionally, a compilation of all the selected features per nodes, edges, and graphs is presented in Table 8.2. Node features primarily relate to pin characteristics that influence the slews and the capacitance load, such as node type, bit width, and the number of input and output connections. The node type attribute

Table 7.1.: Features per nodes, edges, and graph.

Type	Feature	Size	Type	Description
Node	Node type	23	`bin`	One-hot-encoded type of the node
	# Inputs	1	`int`	Number of input edges to the node
	# Outputs	1	`int`	Number of output edges of the node
	Bit width	1	`int`	Bit width of the pin
	Depth level	1	`int`	Level of the node in the DAG
	PI/PO flag	3	`bin`	One-hot-encoded type of pin
	Edge inverter	1	`bin`	Whether a node inverts the timing arcs
Edge	Input slew	4	`float`	Transition time from the input pin
Graph	# Nodes	1	`int`	Total number of nodes
	# Ideal wires	1	`int`	Number of edges with zero delay
	Depth	1	`int`	Max. Distance from root to end node

Table 7.2.: Ground-truth labels or objectives per edges and graph.

Type	Label	Size	Type	Description
Edge	Output slew	4	`float`	Transition time of the output pin
	Delay	4	`float`	Time from input to output pin
Graph	Maximum AT	2	`float`	Max. AT for rising and falling edge

specifies the function of the node within the graph, whether it serves as the root or sink node, a constant, a primary port, or one of the nineteen supported logic gates. Furthermore, an edge inverter flag indicates whether a component inverts the timing arcs, as done by logic gates such as NOT, NAND, and XOR. An additional node feature is introduced to identify the pin type, resulting in a one-hot encoded vector of three elements capable of identifying whether the node functions as a primary input, output, or regular pin. Moreover, edges are annotated only with input slews as features. Lastly, the graph features are designed to encapsulate the complexity of the design, including the total number of nodes, ideal wires, and the maximum distance between the root and sink nodes.

Table 7.2 lists the ground-truth labels or prediction objectives per each edge and the complete graph, which are directly extracted from the timing reports. At the edge level, the regression task is oriented towards predicting the maximum and minimum output slews and delays for both rising and falling timing arcs. On the other hand, for the graph-level task, the regression objectives are the maximum ATs of a design for both rising and falling timing arcs. These labels per edge and graph are mapped to numerical vectors $\mathbf{y}^{\mathcal{E}}$ and $\mathbf{y}^{\mathcal{G}}$ for further processing. Tables 8.2 and 7.2 show that features and ground-truth labels exhibit distinct sizes and data types. Hence, preprocessing becomes imperative. Categorical features are encoded using a one-hot representation, while the remaining features and labels are scaled within the range of $[0, 1]$.

7.4. GNN-based Predictors

The third step in the flow described in Figure 7.2 is the selection of appropriate ML architectures. This chapter uses GNNs to encode the featured graphs and two MLP networks for the

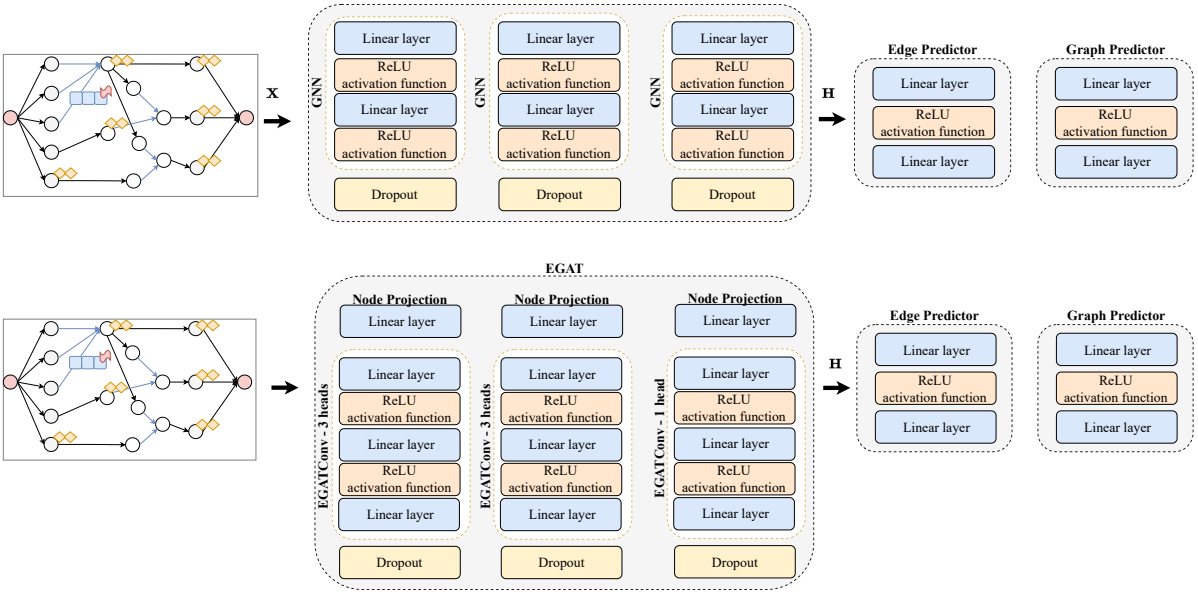


Figure 7.3.: Implemented GNN-based models for the pin-to-pin delay estimation and the maximum AT.

final regression tasks.

As recognized in [135], most GNN architectures do not support multidimensional edge features. To address this limitation, this thesis uses two different convolutional GNNs, namely a modified **GCN** [150] and an **Edge-featured Graph Attention Network (EGAT)** [154]. Figure 7.3 provides an overview of the implemented EGAT and GCN encoding the input graphs. Further details of the extended GCN, presented in Section 6.4.2.1, are applied to the pin-to-pin delay estimation in Chapter 6. At its core, this model represents a generalization of convolution to graphs and is extended so that the node embeddings are updated, also considering the embeddings of the edges. Similarly, EGAT supports multidimensional edge features per definition and generalizes the attention mechanism [155] to graphs. While the extended GCN aggregates information from all neighbors equally, the EGAT evaluates the relevance of each neighbor to a given target node. In this chapter, a GCN is chosen for its simplicity, while an EGAT is employed due to its ability to detect relevant pin-to-pin connections or edges during feature aggregation.

7.4.1. Graph Encoding

This section describes the implemented GCN and EGAT models in Figure 7.3. The resultant lower-dimensional vectors per node, denoted as $\mathbf{h}$, serve as inputs to edge and graph predictors for estimating delays, slews, and ATs.

The **GCN** architecture was explained in detail in Section 6.4.2.1. The implemented GCN model in this chapter is depicted in Figure 7.3a and leverages three GCN and dropout layers to prevent overfitting. In essence, a GCN layer takes node and edge features as input to create a vector m,

which is then used to update the node embedding vectors $\mathbf{h}$ following the principles outlined in Equation 6.1. Each GCN layer consists of two linear layers. For every edge (u, v), the embeddings of the source node $\mathbf{h}_u$ and target node $\mathbf{h}_v$ are concatenated with the embeddings of the edge $\mathbf{he}_{(u,v)}$. This concatenated vector undergoes processing through the first linear layer with ReLU [149] activation function to form a message $\mathbf{m}$. The resulting message is concatenated with the node embeddings and fed again through the second linear layer. The other two GCN layers follow the same updating rule while reducing the dimensionality of the embedding vector $\mathbf{h}$ from 31 to a final size of 15.

The **EGAT** model is an extension of the graph attention network [156], explicitly incorporating support for edge features. The computation flow of EGAT encompasses four primary steps. Initially, the node embeddings are transformed using a linear layer so that $\mathbf{h}_u^{l+1} = \mathbf{W_1}\mathbf{h}_u^l + b$, where $\mathbf{W_1}$ and b are learnable parameters. For each edge connecting a source node u and a target node v, the embedding vector is updated using the current edge embeddings $\mathbf{he_{(u,v)}}$ alongside the node embeddings $\mathbf{h}_u$ and $\mathbf{h}_v$:

$$\mathbf{he}_{(u,v)}^{l+1} = LeakyReLU(\mathbf{W_2}\ [\mathbf{h}_u^l || \mathbf{he}_{(u,v)}^l || \mathbf{h}_v^l]), \tag{7.1}$$

where $\mathbf{W_2}$ represents a learnable matrix, the initial edge embeddings are the edge features denoted as $\mathbf{he}_{\mathbf{(u,v)}}^{\mathbf{(l=0)}} = \mathbf{z}_{(u,v)}$, and $(.||.)$ symbolizes the concatenation operation. Subsequently, the updated edge embedding is employed to calculate a normalized attention score α:

$$\alpha_{(u,v)}^{l+1} = Softmax(\mathbf{W_3}\mathbf{he}_{(u,v)}^{l+1}), \tag{7.2}$$

where $\mathbf{W_3}$ is another trainable matrix. Finally, the attention score α, the neighborhood $\mathcal{N}(\cdot)$ of the nodes, and a non-linear function $\sigma(\cdot)$ are used to update the node embedding vectors:

$$\mathbf{h}_u^{l+1} = \sigma(\sum_{v \in \mathcal{N}(u)} \alpha_{(u,v)}^{l+1} \mathbf{h}_v^l) \tag{7.3}$$

The EGAT-based model is structured with three EGATConv and dropout layers. The computation of the hidden vectors $\mathbf{h}$ in each EGAT layer follows Equations 7.1, 7.2, and 7.3. The first two EGATConv layers employ three attention heads, where the output of each head is concatenated to a new embedding vector and passed through a dropout layer. The final EGAT layer consists of one attention head, and its output is directly connected to the edge and graph predictors.

7.4.2. Edge and Graph Predictor

In contrast to the approach adopted in Chapter 6, where the embedding matrix H serves as the input to one single MLP network, this chapter diverges by employing two separate MLPs trained jointly for edge and graph-level predictions.

The edge predictor responsible for forecasting edge delay and output slew values is built using a two-layer MLP network. This predictor concatenates the source $\mathbf{h}_u$ and destination $\mathbf{h}_v$ embedding vectors for all edges (u, v). The resulting concatenated vector is transformed through

a linear layer with ReLU activation, followed by a second linear layer that maps the resulting vector to a vector of edge labels or regression objectives. As listed in Table 8.2, the objectives are vectors of size eight.

The graph predictor utilizes an MLP network with two linear layers. It takes the graph-level features **g** as inputs and concatenates them with the node embedding vectors **h**. Since the number of nodes per graph is variable, a *readout* mean operation is needed to consolidate all node embeddings into a single vector per graph. The output linear layer comprises two neurons for predicting the maximum AT for rising and falling edges.

7.5. Training

After the circuit generation, synthesis, and collection of the featured graphs, the collected 100K graphs are split into training (70%), testing (20%), and validation (10%) sets. Algorithm 1 describes the implemented models in Figure 7.3 and their main parts for encoding (Lines 6-21), edge (Line 23), and graph prediction (Line 25). Moreover, it also shows the training steps followed. The first step is to initialize the embedding vectors of the first layers with the feature vectors extracted in the GML file. Afterward, the graph encoding follows the GCN or EGAT computation. For both cases, the number of layers L is set to three. After configuring the GNN models, two MLPs are implemented for the edge and graph-level regressions. Each MLP uses a separate loss function. The MAE in Equation 3.7 is leveraged to compute the *edge loss* in Equation 7.4 and the graph loss in Equation 7.5:

$$\mathcal{L}_{edge} = \frac{1}{|\mathcal{E}| * D} \sum_{i=1}^{|\mathcal{E}|} \sum_{j=1}^{D} |\mathbf{y}^{\mathcal{E}}_{(i,j)} - \hat{\mathbf{y}}^{\mathcal{E}}_{(i,j)}|, \tag{7.4}$$

$$\mathcal{L}_{graph} = \frac{1}{D} \sum_{i=1}^{D} |\mathbf{y}^{\mathcal{G}}_{i} - \hat{\mathbf{y}}^{\mathcal{G}}_{i}|, \tag{7.5}$$

where $\hat{\mathbf{y}}^{\mathcal{E}}$ is the vector of predicted slews and delays, $|\mathcal{E}|$ is the total number of edges connecting input and output pins of the same component, $\hat{\mathbf{y}}^{\mathcal{G}}$ is the vector of predicted ATs, and D is the size of the predicted vectors, i.e., $D = 8$ and $D = 2$ for both equations, respectively. Both MLPs are train jointly using a composed loss. The latter is called the *total loss* and is calculated as the sum of both losses in Equations 7.4 and 7.5, i.e., $\mathcal{L}_{total} = \mathcal{L}_{edge} + \mathcal{L}_{graph}$. The total loss is minimized during training using an optimizer such as Stochastic Gradient Descent (SGD).

Moreover, a hyperparameter search determines the optimal parameter values for the learning rate η, momentum β, and batch size b_s. The best hyperparameter values for each model architecture are identified and documented in Lines 11 and 21 of Algorithm 1. Subsequently, the models are trained on an NVIDIA® Tesla® P4 GPU for 100 and 200 epochs, consuming one and seven hours, respectively. This variation in training times can be attributed to the disparity in their number of trainable parameters of 15K for the GCN and 24K for the EGAT model. The pre-trained models are initially evaluated on the test dataset, with the results summarized in Table 7.3 for both models and tasks. Results show that the performance of both models

Algorithm 1 Training flow of GCN- and EGAT-based models.

Input: $\mathcal{G}$: featured DAGs $\{\mathcal{V}, \mathcal{E}, \mathbf{X}, \mathbf{Z}, \mathbf{g}\}$, T: model type, $L = 3$: number of layers, σ: ReLU activation
Output: Models parameters $\mathbf{W}_{1,2,3}$, $b_{1,2,3}$, $\theta_{1,2}$
η: learning rate, β: momentum, b_s: batch size,
$\mathbf{h}, \mathbf{he}$: embedding vectors ▷ Variables
$\mathbf{h}_u \leftarrow \mathbf{x}_u$, $\mathbf{he}_u \leftarrow \mathbf{z}_u$ ▷ Graph encoding
if T == "GCN" **then**
  for $l \leftarrow 1$ to L **do**
    $\mathbf{m}_u \leftarrow \sigma(\mathbf{W_1}(\mathbf{h}_u^l || \mathbf{he}_{(u,v)}^l || \mathbf{h}_v^l) + b_1)$
    $\mathbf{h}_u^{l+1} \leftarrow dropout(\sigma(\mathbf{W_2}(\mathbf{m}_u || \mathbf{h}_u^l) + b_2)$
  end for
  Set $\eta \leftarrow 0.006$, $\beta \leftarrow 0.9$, $b_s \leftarrow 16$
else if T == "EGAT" **then**
  for $l \leftarrow 1$ to L **do**
    $\mathbf{h}_u \leftarrow \mathbf{W_3}\mathbf{h}_u^l + b_3$
    Compute $\mathbf{he}_u^{l+1}$ via Equation 7.1
    Compute α via Equation 7.2
    Get $\mathbf{h}_u^{l+1}$ via Equation 7.3
    $\mathbf{h}_u^{l+1} \leftarrow Dropout(\mathbf{h}_u^{l+1})$
    $\mathbf{he}_u^{l+1} \leftarrow Dropout(\mathbf{he}_u^{l+1})$
  end for
  Set $\eta \leftarrow 0.009$, $\beta \leftarrow 0.95$, $b_s \leftarrow 8$
end if
$\hat{\mathbf{y}}^{\mathcal{E}} \leftarrow \text{MLP}(\theta_1 \mid \mathbf{h}_u^L || \mathbf{h}_v^L),\ \forall (u,v) \in \mathcal{E}$ ▷ edge predictor
$\hat{\mathbf{y}}^{\mathcal{G}} \leftarrow \text{MLP}(\theta_2 \mid \frac{1}{|\mathcal{V}|} \sum_{u=1}^{|\mathcal{V}|} \mathbf{h}_u^L || \mathbf{g}),\ \forall u \in \mathcal{V}$ ▷ graph predictor
Compute $\mathcal{L}_{total}$ adding Equations 7.4 and 7.5
Set optimizer ← stochastic gradient descent
Minimize $\mathcal{L}_{total}$ using optimizer, η and β

Table 7.3.: Training metrics for both models and both regression tasks at edge and graph level.

Model	**Task**	R^2	MAE ns
GCN	Edge	0.97	0.011
	Graph	0.91	0.016
EGAT	Edge	0.99	0.008
	Graph	0.91	0.002

is comparable, with an R^2 higher than 0.97. Additionally, the edge predictions consistently exhibit higher accuracy, which aligns with expectations given the greater complexity of AT estimation, reliant on the timing of all components along the entire path.

7.6. Summary

This chapter draws inspiration from state-of-the-art works [135], [136] jointly estimating delay and slews of pin connections and global timing metrics such as ATs. Remarkably, this chapter estimates the pin-to-pin delays and slews and the maximum ATs of an RTL IR. The GCN architecture proposed initially in Chapter 6 is further employed. For comparison, an EGAT model is implemented as it distinguishes relevant nodes and edges in the neighborhood and considers, per definition, multidimensional edge features. The embedding vectors resulting from the GCN or EGAT models are fed into two MLPs predicting delays and slews per edge

and ATs per graph. Notably, the GNNs and the two MLPs are trained jointly utilizing a composed loss.

In contrast to the methodologies reviewed in Chapter 4, this approach is the first to input RTL IRs and simultaneously estimate post-logic synthesis values for pin delays and ATs. Like this work, some approaches estimate component delays and slews and then propagate them through the timing paths to obtain the maximum ATs of a design [135], [136]. For example, Ye et al. [135] present a GNN for path-based timing analysis. They propose a three-step flow: GNN encoding, cell slew and delay prediction, and critical path AT calculation. The resulting encoding vectors are input to an MLP network estimating the path-based cell slew and delay. The AT is the cumulative addition of the predicted cell delays and wire delays obtained through graph-based analysis. Guo et al. [136] present an end-to-end timing prediction model that maps pre-routing netlists to DAGs and estimates global post-routing timing metrics. Their model targets two tasks: the cell delay and slew estimation and the AT and slack prediction. The complete model is trained by accumulating the losses of each task. In contrast to [135] and like [136], this work uses ML models to estimate the ATs instead of aggregating or adding the predicted component delays.

Furthermore, approaches presented in both [136] and [135] aim at optimizing designs during physical synthesis. Thus, they build input graphs from pre-routing gate-level netlists, where the RTLs microarchitecture is fixed. The main contribution of this chapter is on the use of ML to predict not only pin but global timing metrics from a very early design stage. However, it is essential to note that the graph predictor introduced in this chapter is confined to predicting only the maximum values among all primary outputs of a design. While this can aid in comparing different RTL variants, it does not facilitate the identification of critical components and blocks that could serve as candidates for microarchitecture optimizations. Moreover, relying on ML for metric prediction confines the regression space to the distribution utilized during training. To address this limitation, Ye et al. [135] propose estimating and aggregating pin delays to compute path ATs. This approach ensures that the ML regression constrains solely the space of the predicted pin delays and not the computed ATs. This distinction is beneficial as the ranges of pin delays are less diverse among different components and circuits. In contrast, ATs depend on the entire design and scale with the complexity of the path.

8. Fake Timer: ML-Enabled Block-Based Static Timing Analysis

This thesis employs ML to approximate the timing analysis results of a design at earlier stages of the design flow. In Chapter 6, the estimation process for delays and slews of pin-to-pin connections in a RTL IR is discussed. While this estimation provides insight into the critical pins of a single component, it fails to offer an overarching timing metric to identify critical paths, regions, or blocks of a design. To address this limitation, Chapter 7 proposes using GNNs to predict the maximum ATs at design endpoints. The estimated ATs correspond to the worst-case path delay in a design and can be employed to compare different RTL microarchitectures or HDL variants. However, this approach does not allow pinpointing critical blocks within the RTL, which would open the door to design optimizations at the early stages of the design flow.

Related works reviewed in Chapter 4 leverage ML techniques to predict timing metrics mostly in a path-based approach. Like Chapter 7, they aim to predict global design metrics that allow the comparison of the critical path of distinctive design variants. Conversely, this chapter aims to annotate timing metrics for each component, and by doing so, it detects critical parts of the design. The main goal is not the comparison of timing metrics among designs but instead the guidance of microarchitecture corrections within a single design. To achieve this, a close integration of ML models, classical STA, and optimization methods is proposed based on annotating RTL blocks with accurate timing estimation. The following sections define the problem and the key steps to address it using the ML methods from Chapter 3 and the lessons learned from Chapters 6 and 7.

8.1. Problem Statement

Given an RTL IR composed of logic components, early estimations of global timing metrics such as ATs, RATs, and slacks are needed to compare different RTL variants, pinpoint critical blocks in the design, and guide corrective measures. Such global metrics require the estimation of pin-to-pin delay and slew values. For instance, given the RTL IR in Figure 8.1, delay and slew values should be predicted for every pin-to-pin connection highlighted in blue, and accurate ATs, RATs, and slacks should be annotated to every pin. Pins with the same maximum slack value correspond to the critical path, and the highest AT is the worst-case path delay.

Back-annotating realistic timing from gate-level timing reports to components of an RTL IR is challenging, especially when the gate-level netlist is obtained through flattened synthesis. In that scenario, combinatorial blocks are merged into a single module, as depicted in Figure 5.5, and multiple cross-boundary Boolean optimizations are applied. Consequently, RTL boundaries may no longer be discernible in the netlists, with only flip-flops or registers, latches, and

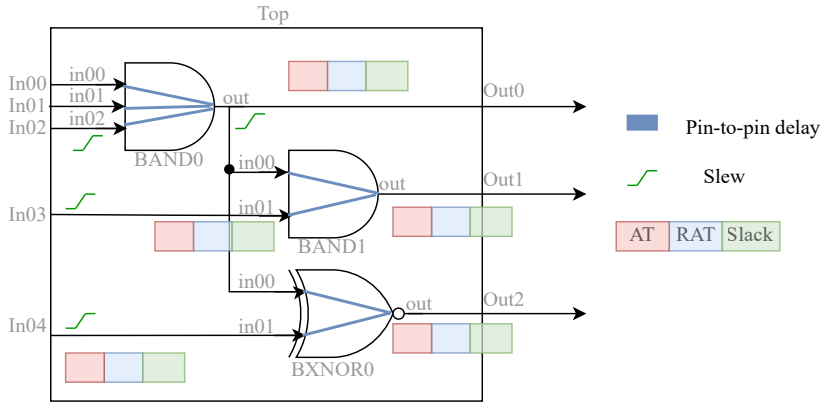


Figure 8.1.: RTL IR of a combinatorial circuit with pin-to-pin delays and ATs, RATs, and slacks to estimate.

primary ports remaining identifiable. To back-annotate timing from a flattened netlist to the RTL blocks, a thorough analysis is required to map combinatorial clouds between identifiable components to specific RTL blocks. This analysis is time-consuming as it should be performed for all paths passing through all design components. Moreover, timing reports are path-based, as shown in Figure 5.8. They do not provide information about the worst RAT or slack per pin or component. Instead, they list the pin-to-pin delays, slews, and the ATs for each pin but within a given path defined by a start and endpoint. Only at the path's endpoints, a slack measure is calculated based on the RATs. Furthermore, timing reports find the best and critical paths but not the specific critical blocks from where the signal started getting late or critical across all the possible paths.

To overcome this, this chapter proposes a novel timing estimation flow called *Fake Timer*. Fake Timer annotates estimated pin-to-pin delays, ATs, RATs, and slacks to each block of an RTL design. Initially, the estimated metrics are inaccurate as they do not account for cross-boundary optimizations possible during flattened synthesis. To consider timing from flattened netlists, Fake Timer *corrects or fakes* the estimated delays to minimize the difference between the computed ATs and the results after flattened synthesis. Using corrected delays and ATs, RATs, and slacks are computed, ensuring realistic timing is back-annotated to RTL blocks, even if their boundaries do not appear in the synthesized netlist. The Fake Timer methodology is composed of three key stages, as illustrated in Figure 8.2: (i) the prediction of pin-to-pin delays and slews, using a model per component type; (ii) the block-based STA using ML-predicted delays and slews, and (iii) the minimization of the computed ATs concerning the results obtained from flattened synthesis. The following sections provide details of the implementation of these three main steps.

8.2. Consolidation of Pin-to-Pin Delay and Slew Models

Like Chapter 6, Fake Timer employs NNs to predict pin-to-pin delays and slews of primitive components within an RTL IR. However, this chapter opts for simple MLP networks and reduces the complexity of the problem by training one model per component type. The decision

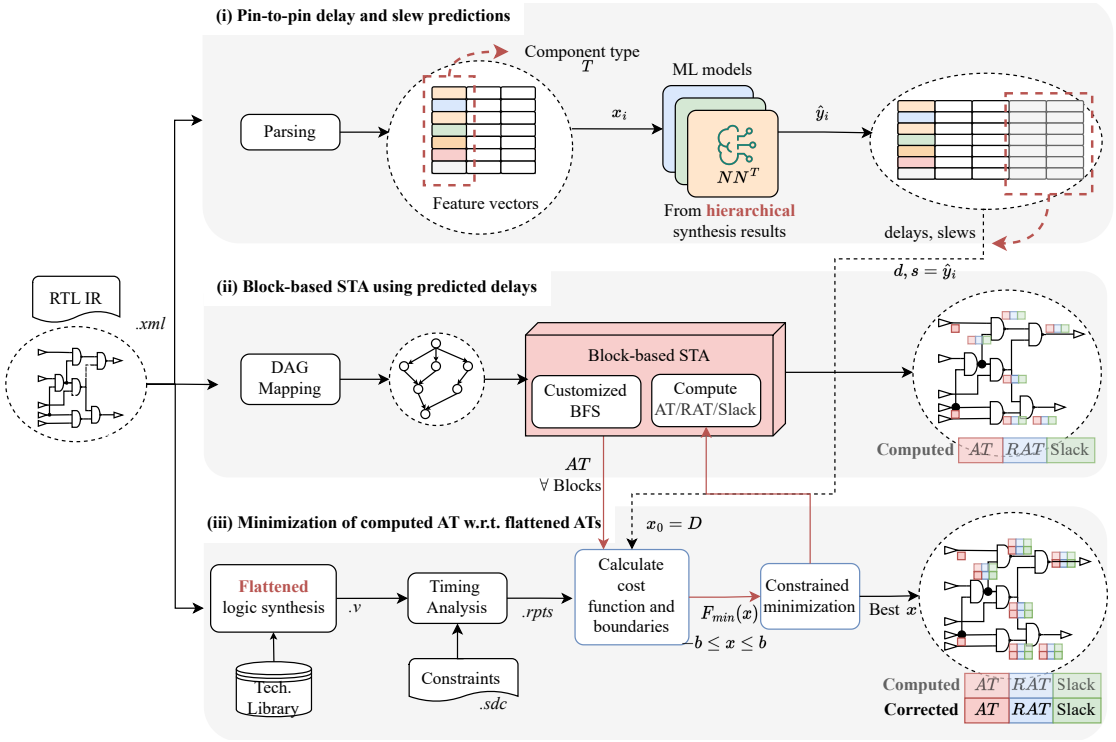


Figure 8.2.: Fake Timer flow proposed.

not to employ GNN-based models is influenced by their limited interpretability and generalization issues with unseen complex designs with long paths, as demonstrated in the findings of Chapter 7. Learning from the outcomes of Chapter 6, the final Fake Timer flow incorporates four significant differences in the delay and slew estimation models. Figure 8.3 provides an overview of the main distinctions compared to the flow in Chapter 6. Specifically, the generation of training circuits is done using a more restricted set of random specifications. Additionally, timing reports are obtained from eighty-one parallel runs with different constraints. Lastly, the selection of features, model architectures, and model training are done separately per each component type.

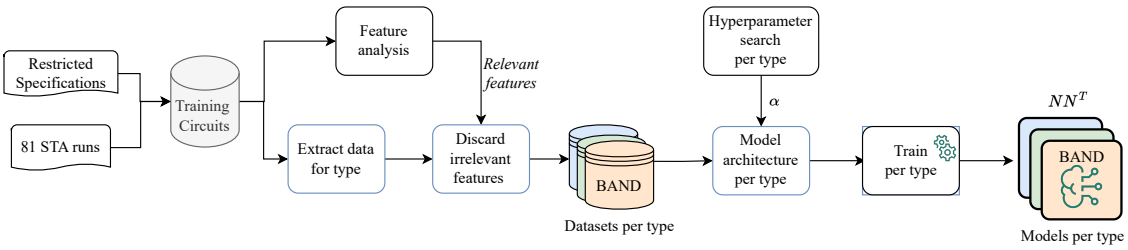


Figure 8.3.: Dataset collection and model training based on the component type.

Table 8.1.: Total number of random specifications per component type used for the Fake Timer flow.

Comp. Category	Comp. Type	Total No Fan-out	Total Fan-out
Logic	BAND	63	2079
	BNAND	1	1
	BOR	96	1440
	BNOR	1	2
	BXOR	27	135
	LAND	21	315
	LNAND	1	1
	LOR	27	216
	LXOR	2	2
	RAND	30	30
	RNAND	1	2
	ROR	29	174
	RNOR	1	1
	RXOR	7	14
	NOT	11	66
Comparator	EQ	17	238
	NEQ	8	16
	GTEQ	14	42
	GT	14	42
	LTEQ	6	18
	LT	12	48
Multiplexing	MUX	952	22848
	DEMUX	192	768
Arithmetic	HWPLUS	69	345
	HWMINUS	19	95
	HWMUL	16	8
Total	30583	1637	28946

8.2.1. Training Circuits and Timing Reports

Leveraging the data generation flow introduced in Chapter 5, circuit specifications are generated for each component type. The objective is to produce cleaner and more focused datasets by confining the RTL-related features to meaningful and commonly utilized configurations while encompassing a wider range of timing-related features.

In contrast to the data generation approach in Chapter 6, which utilizes *general* specifications, this chapter employs *restricted* specifications as outlined in Table A.2. These configurations are carefully chosen based on configurations employed in established hardware modules such as hardware adders, multipliers, RISC-V cores, and SoC designs. This strategy avoids the generation of many valid but unnecessary configurations. Table 8.1 provides an overview of the total number of designs generated per component type, with and without fan-out. Designs without fan-outs encompass single components without additional gates connected to the component's output pin, while designs with fan-outs involve extra gates connected to the output pin. In total, 30.6K designs are automatically generated. These are less than the circuits generated for DS-I, DS-II, and DS-III. The reduction of generated circuits allows the model to focus on relevant RTL-related features.

For every training circuit and its gate-level representation, STA is executed in parallel eighty-one times, following the synthesis flow outlined in Figure 5.6b. Each run employs different input slews and delays for the primary inputs ranging between [0, 1] and [0, 2], respectively. Despite the reduction in the number of generated circuits due to the restricted set of specifications, the parallel synthesis flow increases the coverage of timing conditions. Consequently, eighty-one distinct samples are obtained from each circuit, represented by identical RTL-related features but with varying timing features and ground-truth labels. During both training and evaluation, the remaining timing constraints specified in Table 5.1 remain constant.

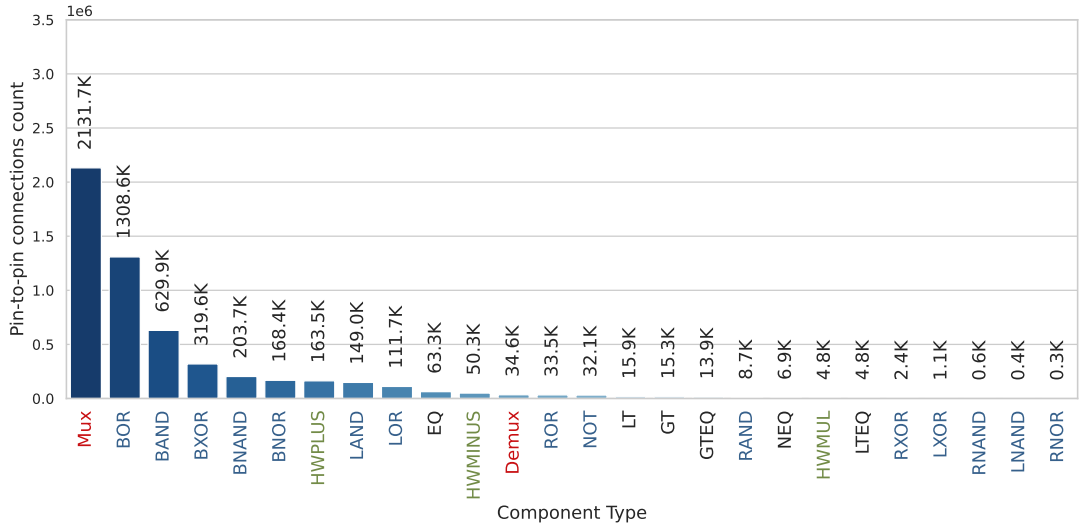


Figure 8.4.: Number of samples or pin-to-pin connection counts in the tabular datasets corresponding to each component type.

8.2.2. Datasets per Component Type

In this chapter, exclusively tabular datasets are built due to the higher explainability of MLPs compared to GNNs. As depicted in Figure 6.2, each pin-to-pin connection within the components of the generated MoDs is mapped to a sample row in the dataset. The histogram illustrated in Figure 8.4 shows the distribution of the 26 component types and their respective number of pin-to-pin connections, representing the samples in the training dataset. In the figure, component types with the same color are grouped in the same category. The histogram also reveals the prevalence of the MUX type at the top, as its input count can vary from two to 64, leading to a corresponding high variation in the number of pin-to-pin connections per each MUX. Conversely, components like the RNOR, LNAND, and RNAND exhibit a smaller quantity of generated MoDs, attributed to the limited range of their selected configurations listed in Table A.2.

The feature selection in this chapter draws inspiration from the features utilized for DS-I and DS-II, which are detailed in Table 6.2. However, this chapter modifies the features related to the pin bit widths, incorporating the size of the current input and output pin with additional bit-width statistical measures, such as the sum, average, minimum, and maximum among all input pins. These refined features for each pin-to-pin connection are enumerated in Table 8.2. Furthermore, the objectives used for DS-I and DS-II are employed in this chapter too and listed in Table 6.2 again for completeness.

One of the main learnings from Chapter 6 is the observation that augmenting the dataset with additional component types increases its complexity and diminishes the correlation among pairs of features. Consequently, the correlation among features is, on average, stronger for the DS-I dataset and lower for DS-II, as depicted in Figure 6.4. For the dataset generated in this chapter, Figure 8.5 shows the correlation between the objectives and the features for

Table 8.2.: Features and objectives for each pin-to-pin connection.

Features	Symbol	Size	Type	Description
Component type	x_0	26	`bin`	One hot vector of 26 possible types
Number of pins	x_1, x_2	2	`int`	Total number of input and output pins
Input number	x_3	1	`int`	Current input pin position
Bit width	x_4, x_5	2	`int`	The bit size of the current input and output pins
Connections	x_6, x_7	2	`int`	Connection to the current pin and output
Bit width stats.	x_8, x_9, x_{10}, x_{11}	4	`int, float, int, int`	Sum, avg., max., and min. bit widths of all inputs
Input slew	x_{12}	4	`float`	Transition times for current pin
Objectives	**Symbol**	**Size**	**Type**	**Description**
Output slew	y_0	4	`float`	Transition time for output pin
Delay	y_1	4	`float`	The time between the current input and output

all component types, excluding the feature component type, to keep the heatmap simple and condensed. In the heatmap, darker colors signify that as one variable increases, the other tends to decrease. Notably, certain features exhibit high correlations with each other as expected, such as the number of inputs x_1 and the statistics for the input bit widths, including the sum x_8, the average x_9, the maximum x_{10}, and the minimum values x_{11}. Analyzing the objectives y_0 and y_1, it becomes evident that their correlation with each other is stronger than with other features. Moreover, the feature with the highest correlation to the objectives is the fan-out x_7. However, the correlation for the remaining features and objectives hovers between 0.2 and 0, indicating a lack of significant correlation. This weak correlation suggests that the dataset is overly complex, and its features are not discriminative or correlated enough. Therefore, an analysis of the feature selection is needed per each component type.

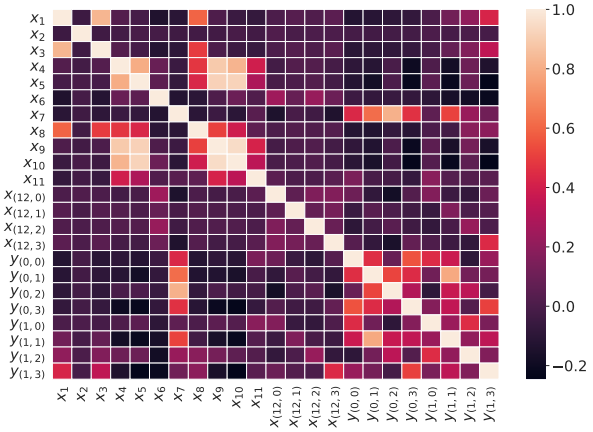
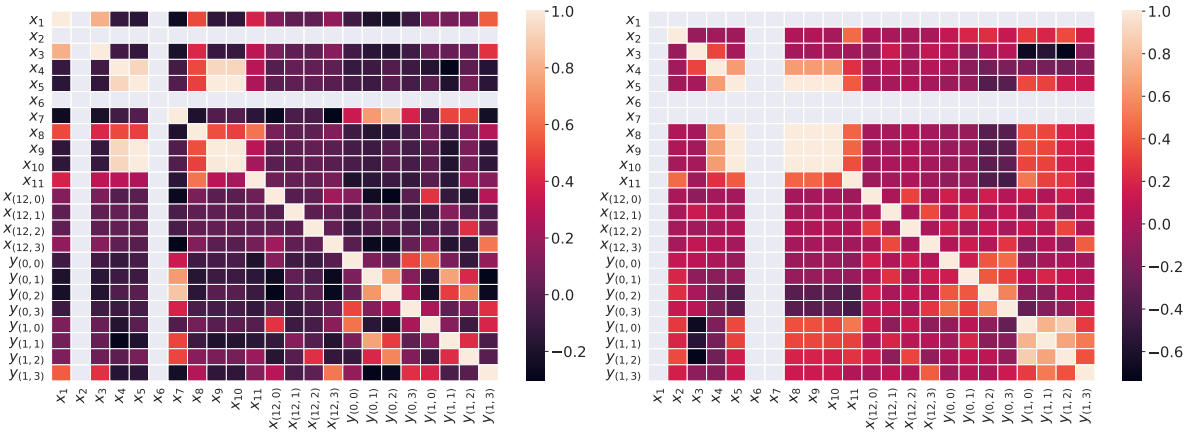


Figure 8.5.: Spearman's correlation matrix of all collected samples considering features and objectives of Table 8.2

Different component types exhibit different configurations and timing behaviors, leading to the relevance of extracting specific features for each type. For instance, the number of inputs in a MUX can vary from two to N, making this feature particularly informative. In contrast, DEMUXes and comparators always have two inputs, rendering the number of inputs as a feature redundant for these component types. Figure 8.6 illustrates the heatmaps for the



(a) Type MUX (b) Type DEMUX

Figure 8.6.: Heatmaps for two different component types

MUX and DEMUX component types, respectively. For the latter, the features number of inputs x_1 and fan-in x_6 do not provide significant information as expected. Conversely, the remaining features exhibit a stronger correlation with the objectives. In the heatmap for the MUX type, portrayed in Figure 8.6a, the overall pairwise correlation is lower. This could be attributed to the considerable number of different possible configurations, resulting in a larger number of training circuits with lower feature correlation.

Based on the SRCC results, specific features are selectively excluded from the final feature vectors $\mathbf{x}$ depending on the component type. For example, the number of outputs x_2 is preserved only for the DEMUX type, while for this type, the number of inputs x_1 is instead dropped. In the case of comparators, the number of inputs and outputs are fixed to two and one, respectively; thus, these features become irrelevant. Similar to the NOT type, which always has one single input, the current input pin position x_3 loses its relevance. Consequently, individual tabular datasets are constructed for each component type. Each row represents a pin-to-pin connection, as illustrated in Figure 6.2a, while the columns are a subset of the features listed in Table 8.2 and depend on the component type associated with the given pin-to-pin connection.

8.2.3. Models per Component Type

This chapter deviates from the approach of Chapter 6 by training one model for each component type T, as shown in Figure 8.3. This strategic shift is based on recognizing that different component types possess unique features and timing behaviors, necessitating individualized datasets, and training configurations. Thus, different features and datasets are required, as mentioned in the previous section. Consequently, 26 MLP models are trained, i.e., one model for each component type, varying in the number of hidden layers utilized (between two and five). The first layer contains a number of neurons equal to the number of features for the respective component type. For the BAND, the total number of features is 14, while for the comparators, it is 13, and for the NOT gate, it is 12. The number of neurons in the output

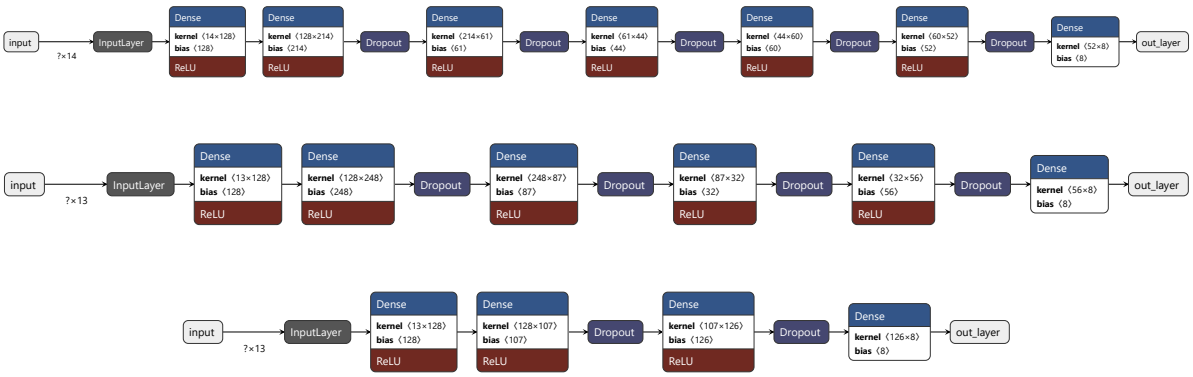


Figure 8.7.: Models for three component types. From top to bottom: HWMUL, EQ, and GT.

layer is always eight, where the first four neurons predict the slew and the last four the delay. Four values are predicted, covering maximum and minimum delay, as well as rising and falling edges.

Afterward, a hyperparameter search is conducted to determine the optimal number of layers, neurons per hidden layer, optimizer, loss function, and dropout and learning rates. The selected loss function is one of the possible error regression metrics, such as the MSE, the MAE, and the Mean Squared Logarithmic Error (MSLE). The outcomes of the hyperparameter search for each component type are presented in Table A.1. Furthermore, the selected architectures for certain component types, including EQ, NOT, and HWMUL, are shown in Figure 8.7, which are generated using Netron [157]. The individual MLP models for each component type are trained over 50 epochs, utilizing their respective optimizer and loss function. Each model is trained with 70% of the collected tabular dataset, with 20% allocated for hyperparameter search and the remaining 10% reserved for testing.

Finally, after the 26 MLP models are trained with their respective configurations and datasets, they can be evaluated over single components or entire RTL IRs. In the last use case, every component of the IR is assessed. The component type is identified, and for each connection between an input and the output pin of the component, the corresponding model is retrieved. Subsequently, the inference is conducted using the feature vectors that describe the specific pin-to-pin connection. An example will be provided in the following section to illustrate this process.

Running Example:

Given the c17 design from the ISCAS'85 [158] benchmarks, Figure 8.8 shows its MoD or RTL IR. The IR comprises six BNAND components, amounting to a total of 12 pin-to-pin connections, highlighted in red. Additionally, it has five Primary Inputs (PIs) and two Primary Outputs (POs). The ML model specific for the BNAND component type is invoked six times to predict the minimum and maximum delays and slews. As per the definitions in Table 8.2, the output slew is represented by y_0, and the delay is represented by y_1, both of which are vectors of size four, accounting for maximum and minimum, rising and falling edges. For the

sake of simplicity, only the predicted and ground-truth values for the maximum rising $y_1[0]$ and falling edges $y_1[2]$ are listed in Table 8.3. The predicted values or outputs of the model are denoted by $\hat{y0}$ and $\hat{y1}$. Averaging the error metrics across all 12 pins, the slew prediction reaches an MSE of 0.06 ps^2, an MAE of 0.019 ns, and a ME of 0.043 ns. Regarding the delay prediction, the MSE is 0.95 ps^2, the MAE is 0.03 ns, and the ME is 0.061 ns.

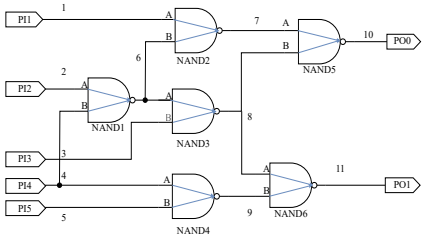


Figure 8.8.: C17 RTL IR.

Component	Input	$y0[0]$	$\hat{y}0[0]$	$y1[0]$	$\hat{y}1[0]$	$y1[2]$	$\hat{y}1[2]$
NAND1	A	0.08	0.07	0.11	0.06	0.07	0.03
	B	0.08	0.06	0.10	0.01	0.07	0.02
NAND2	A	0.06	0.07	0.09	0.07	0.05	0.03
...	...	...	...	...	...	...	...
NAND6	A	0.03	0.07	0.04	0.04	0.03	0.03
	B	0.03	0.06	0.03	0.01	0.02	0.02

Table 8.3.: Pin-to-pin estimations of the c17 benchmark.

8.3. Block-based STA Using ML Predictions

The previous section outlines how the models exclusively predict pin-to-pin connection delays and slews, offering insights into critical pins within a component. However, these predicted values do not provide any conclusion about timing-critical components or paths inside the complete design. To that end, the ATs at the primary inputs should be defined and propagated along the timing graph, where after each component, the delay and slews at the output pin are also accumulated or propagated.

The conventional timing analysis flow is explained in Section 2.3. Commonly, STA tools perform a path-based analysis, i.e., each path of the circuit is analyzed by propagating input slews and approximating the delay through each gate in the timing path. This approximation typically involves interpolation or extrapolation techniques. Each pin is analyzed as many times as it appears in any path. The primary benefit of the path-based analysis is the reduced pessimism, as actual slew and delay values are propagated through each path. Conversely, block-based STA propagates the worst-case scenario slews and delays, not on every path, but block by block. Although this increases pessimism in the results, it is a more computationally efficient method.

The benefits of the block-based algorithm can be summarized in two main points: speed and comprehensive results. Firstly, its efficiency surpasses path-based analysis, as it operates on a block-by-block basis. Particularly in larger designs, path-based analysis becomes computationally intensive as the number of paths increases exponentially with the circuit size. Secondly, the results derived from a block-based approach provide a slack value for each node in the timing graph rather than solely for the endpoints of a circuit. These slack values are invaluable for hardware designers, as they precisely identify areas within the RTL design that could benefit from restructuring or optimization.

This thesis adopts the block-based STA method, initially proposed by Hitchcock et al. [48] and extend it with slew propagation. This thesis diverges from conventional methods that

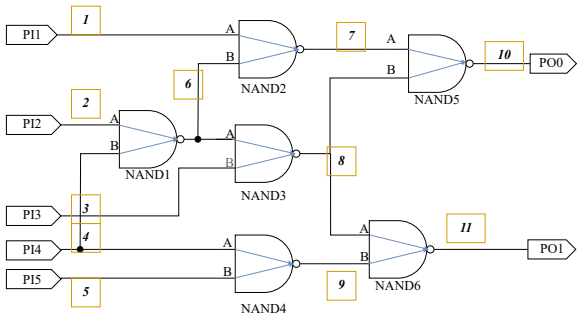
typically depend on hard-coded delay values, or interpolate, or extrapolate the NLDMs for each block. We leverage ML models to predict slews and delays for every input-to-output connection within each component. Using these predicted values, the three steps of the block-based STA approach are conducted: First, a forward propagation step is conducted to compute the ATs. Afterward, a backward pass is executed to determine the RATs. Finally, these calculated values are subtracted from each other to derive the slack values. Further insights into the implementation details for these steps are provided in the following sections.

8.3.1. Graph Traversal

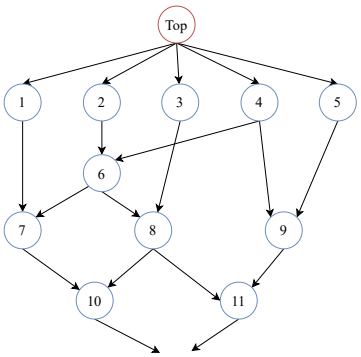
The preliminary steps of timing analysis involve constructing the timing graph representing the RTL IR and determining the appropriate traversal order. Similarly to Chapters 6 and 7, the RTL IRs are mapped to featured DAGs using the pin-node convention displayed in Figure 6.3. In this representation, the primary ports and pins of the circuit represent the nodes $\mathcal{V}$ in the graph. The graph incorporates two nodes, namely a top or root node, and an end or sink node. The edges $\mathcal{E}$ establish connections among nodes, linking primary ports to pins, input to output pins on the same component, or output and input pins of different components. Each edge in the graph holds delays $d(i, j)$ and slews $s(i, j)$ as features, where (i, j) are the source and target nodes connected by the edge $e = (i, j)$. Additionally, an edge contains input and output slews denoted as s^{IN} and s^{OUT}, respectively. Similarly, nodes store RATs, ATs, and slack calculations. The delays and slews for the primary ports are extracted from the timing constraints. In order to support hierarchical designs, extra nodes representing the interfaces and input pins are added. Ultimately, connections between sub-modules are considered ideal wires.

After obtaining the graph representation, the next step is to find a proper order to traverse the graph. A widespread recourse for this is the Breadth-First Search (BFS) algorithm. The BFS algorithm operates in a breadth-oriented manner or level by level, where a level groups nodes that have the same distance to the top node. BFS traverses the graph starting from its top node and explores all the nodes at the current level before moving on to the nodes at the next depth level. Doing so guarantees that it visits all the nodes that are reachable from the starting node. Nevertheless, this search is unsuitable for the block-based timing analysis, as one node should be visited once all its predecessors are visited before [159].

To clarify this concept, let us consider a concrete example. Figure 8.9b shows the DAG for the c17 benchmark. Using the pin node convention, pins and primary ports correspond to nodes. Thus, the pin numbers in the schematic correspond to the node numbers in the graph. The depicted DAG comprises a total of 13 nodes and 16 edges and has five levels. When applying the BFS order to traverse this graph level by level, the resulting search path from the top to the end node is as follows: *1-2-3-4-5-**7**-**6**-8-9-10-11*. This sequence leads to *node 7* being visited before *node 6*. However, computing the AT for *node 7* necessitates the maximum output slews and ATs of its predecessors, i.e., nodes *1* and *6*. Since *node 6* has not been visited at this point, its information is not computed and cannot be propagated to node 7. Therefore, the required traversal order to perform timing analysis over this graph is *1-2-3-4-5-**6**-**7**-8-9-10-11*, where *node 6* is visited before *node 7*.



(a) RTL IR



(b) DAG representation

Figure 8.9.: Example of DAG representation of the c17 benchmark.

To obtain a proper traversal order, a customized BFS is implemented, the steps of which are listed in Algorithm 2. This customized BFS traverses the graph level by level as the standard BFS but visits a node only if all predecessors have already been visited [160]. Thus, it requires the proper initialization of all primary inputs [159]. In detail, Algorithm 2 inputs a list of nodes $\mathcal{N}$ and a list of primary inputs $\mathcal{PI}$ and outputs a list Q containing the traversal order. The algorithm begins by defining the maximum number of levels of the graph, denoted as l_{max}, as the maximum length over all paths of the graph, from the top to the end node. Subsequently, a matrix L of zeros is created to store the queue for each level of the graph. The primary inputs are appended to the first level. The algorithm then iterates through each level from one to l_{max}, exploring the nodes in a breadth-first manner. For each node, it checks whether all of its predecessors have been visited before. In this case, the current node is added to the traversal list Q. Additionally, it identifies the successors of the current node and adds them to the respective level queue. This process continues until all levels have been systematically traversed.

Algorithm 2 Customized BFS.

Inputs: $\mathcal{N}$: List of nodes, $\mathcal{PI}$: List of primary inputs
Outputs: Q: List with traversal order
$l_{max} \leftarrow$ Max. Length of all paths from **top** to **end** node ▷ Get the maximum level of the graph.
$L \leftarrow$ Matrix of zeros of size $l_{max} \times 1$ ▷ Queue for each level of the graph.
$L[1] \leftarrow \mathcal{PI}$ ▷ Append primary inputs.
for $l \leftarrow 1$ to l_{max} **do**
  while $L[l] \neq$ empty **do**
    $n \leftarrow L[l].pop(0)$
    if n not in Q **then**
      if All $p \in$ predecessors(n) in Q **then**
        Append n to Q ▷ Mark n as visited.
      end if
    end if
    for s in successors(n) **do**
      $ls_{max} \leftarrow$ Max. Length of all paths from **top** to **s** ▷ Get the maximum level for s in the graph.
      if s not in $L[ls_{max} - 1]$ **then**
        Append s to $L[ls_{max} - 1]$ ▷ Add s to the queue in the respective level.
      end if
    end for
  end while
end for

8.3.2. Arrival Times

Once the traversal order Q is established, each pin-to-pin connection in the timing graph is analyzed following the steps in Algorithm 3. Initially, all the edges going from the top node to the primary inputs are initialized with their respective delay $d_{(0,PI)}$ and slew $s^{IN}_{(0,PI)}$ vectors. These nodes are marked as visited, and the analysis moves to the succeeding nodes. In cases where an edge connects a component's output pin to another component's input pin, the connection is treated as an ideal wire with a delay of zero and an output slew identical to the input slew (Lines 8-9). For connections linking pins within the same component type T (Lines 10-14), the ML model specific to type T is employed. Here, the tabular features collected from the flow depicted in Figure 8.3 are utilized as input features for the particular MLP with type T. This model then predicts the delay and output slew for that specific connection or edge, as detailed in Section 8.2. After all predecessors of a node are visited, the node's AT can be computed for each timing arc (Line 11) using Equation 8.1.

$$\mathbf{AT}(n) = max_{\forall p \in pred(n)}[\mathbf{AT}(p) + \mathbf{d}(p,n)], \forall n \in \mathcal{V}. \tag{8.1}$$

For the sake of clarity, Algorithm 3 presents a general flow that is agnostic to the specific timing

Algorithm 3 Block-based AT calculation using ML predictions.

1: **Inputs**: $\mathbf{d}_{(0,\text{PI})}$, $\mathbf{s}^{\text{IN}}_{(0,\text{PI})}$: Delays and slews of PIs, Q: Traversal order given by customized BFS
2: **Outputs**: $\mathbf{AT}(n)$ for all nodes $n \in \mathcal{V}$
3: **for each** n in Q **and** n is *visited* **do**
4: **for each** p in $predecessors(n)$ **do**
5: $\mathbf{s}^{\text{IN}} \leftarrow \max_q[\mathbf{s}^{\text{OUT}}(q,p)]$, $\forall q \in predecessors(p)$ ▷ Propagate maximum slew.
6: **if** edge (p,n) is wire **then**
7: $\mathbf{d}(p,n) \leftarrow 0$, $\mathbf{s}^{\text{OUT}}(p,n) \leftarrow \mathbf{s}^{\text{IN}}(p,n)$
8: **else**
9: $T \leftarrow$ type of n ▷ Run inference of NN^T.
10: $\mathbf{x}_e \leftarrow$ features for edge (p,n) from Table 8.2
11: $\mathbf{d}(p,n), \mathbf{s}^{\text{OUT}}(p,n) \leftarrow \text{NN}^T(\mathbf{x}_e, \mathbf{s}^{\text{IN}})$
12: **end if**
13: Mark edge (p,n) as *visited*
14: **if** All p are *visited* **then** ▷ Propagate maximum ATs.
15: Compute $\mathbf{AT}(n)$ via Equation 8.1
16: Mark n as *visited*
17: **end if**
18: **end for each**
19: **end for each**

edges. However, the flow is conducted for the four values rising and falling for maximum and minimum delays. Notably, specific components, such as NOT, XOR, and NAND, are unnate or invert their timing arcs, as indicated by their *timing sense* description from the technology liberty files. In such instances, the AT for rising and falling edges are computed inverting the edges for the delays, i.e., using falling and rising delays, respectively. An example of the results after the AT propagation is presented below.

Running Example:

Figure 8.10 illustrates the predicted pin-to-pin delays and the computed ATs for the c17 design. Although the Fake Timer can predict and compute values for all four timing edges, this

figure exclusively presents the maximum rising edge values. For the sake of simplicity, the computation is done without considering the inverting properties of the NAND gates, and the ATs at the primary inputs are set to the same value of 1.4 ns. Both synthesis and STA should be conducted to evaluate the computed ATs. For comparison, flattened or hierarchical synthesis are performed. From all timing reports, the maximum ATs for each primary output are collected. These are depicted in the grey boxes of Figure 8.10. On average, the computed ATs exhibit an ME of 54 ps and a MAPE of 2.72% compared to the hierarchical results, and an ME of 41 ps and a MAPE of 1.49% regarding flattened synthesis results.

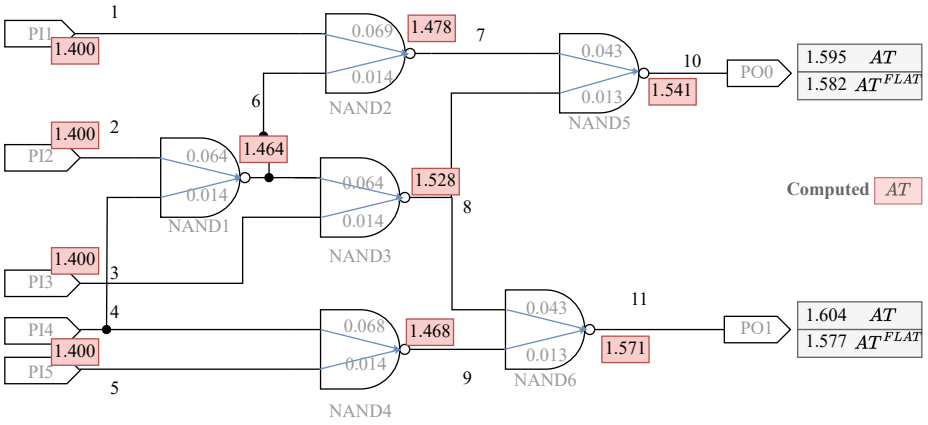


Figure 8.10.: Predicted pin-to-pin delays and computed ATs for the c17 design.

8.3.3. Required Arrival Times

Having calculated the AT and pin-to-pin delays in the forward pass, the next step involves computing the RAT for all nodes in the graph. To that end, the RATs at the primary outputs are calculated and defined based on the given output external delay and the clock configuration as described in Section 2.3. Afterward, the traversal order found using the customized BFS in Algorithm 2 is reversed. The process of back-propagating the RATs from the primary outputs to the inputs is delineated in Algorithm 4. In general, for all successors $s \in successors(n)$, Equation 8.2 is computed.

$$\mathbf{RAT}(n) = min_{\forall s \in succ(n)}[\mathbf{RAT}(s) - \mathbf{d}(n,s)], \forall n \in \mathcal{V} \tag{8.2}$$

Running Example:

Given the output external delays set to 0.8, a clock period T_{clk} equal to 6 ns with an ideal delay of 1 ns and uncertainty of 0.6 ns, the data required time or RAT at both primary outputs was calculated using Equation 2.2 for the maximum rise edge, which is 5.6 ns. The back-propagated RATs are shown in the blue boxes of Figure 8.11.

Algorithm 4 Calculation of block-based required arrival times.

Inputs: Q: traversal order given by customized BFS, PO: List of primary outputs, RAT(PO): Vector of RATs per each PO, D: Matrix of delays for all edges in E
Outputs: $\mathbf{RAT}(n)$ for all nodes $n \in \mathcal{V}$
$Q^R \leftarrow Q$.reverse()
Mark PO as visited nodes
RAT(PO) $\leftarrow$ Input RAT values ▷ RATs for POs are given.
for each n in Q^R **and** n is visited **do**
    for each s in $successors(n)$ **do**
        $RAT_{succ} \leftarrow RAT(s)$
        Delay $\leftarrow D(n, s)$
        $RAT_n[s] \leftarrow RAT_{succ}$ - Delay ▷ Calculate the RATs of n from node s.
    end for each
    $RAT(n) \leftarrow \min(RAT_n[s])\ \forall s \in successors(n)$
    Mark n as visited node
end for each

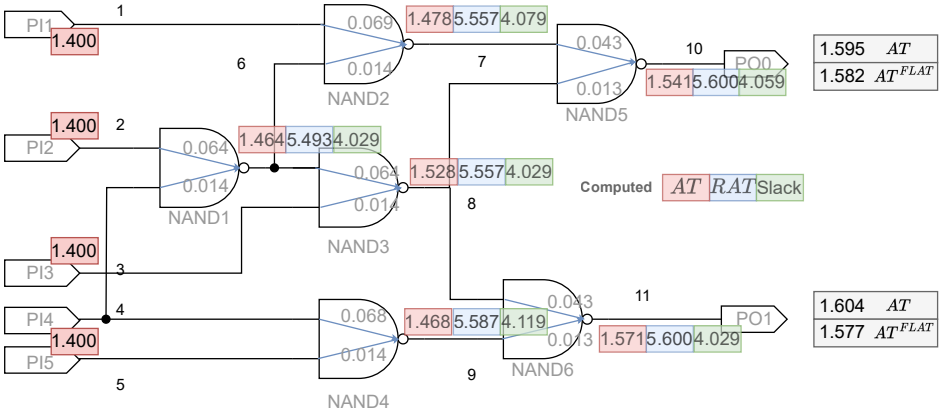


Figure 8.11.: Computed RATs (blue boxes) and slacks (green boxes) for the c17 design.

8.3.4. Slacks

The final step is subtracting the calculated RATs and ATs for every node in the graph. The result of this subtraction is called slack and is calculated as shown in Equation 8.3.

$$\mathbf{Slack}(n) = \mathbf{RAT}(n) - \mathbf{AT}(n), \forall n \in N \tag{8.3}$$

Running Example:

Given the c17 design with computed ATs and RATs, the calculated slacks are shown in the green boxes of Figure 8.11. As all slacks are positive, no critical blocks are found in the RTL IR. However, very high slack values indicate possible optimization room for area. This is the main benefit of the presented approach: estimated yet accurate slacks can be obtained for all pins of an RTL IR without running any tool and allowing optimizations towards area or timing at the early stages of the design flow.

8.4. Correction of Computed Arrival Times

Timing computations in Section 8.3 are based on the ML-predicted delays and slews from Section 8.2. These predictions are obtained from models trained with features and ground-truth values extracted directly from timing reports after hierarchical synthesis. Hierarchical synthesis enhances human readability and debugging and, in this thesis, facilitates the matching of RTL modules to gate-level blocks and parsing their delay and slews. However, it limits cross-boundary optimizations, which are only possible when the design is flattened to a single module. In practical applications, complete flattening of large designs is performed when the tool can manage such complexity. In this thesis, flattening is possible as non-complex designs are targeted, which are only composed of combinatorial sub-blocks. Flattening the design allows for a more realistic timing analysis. However, back-annotating timing from flattened netlists to RTL becomes challenging as the references to RTL blocks are no longer available. Thus, the MLP models in Section 8.2 cannot be trained with flattened results directly.

Even though the computation of timing metrics in Section 8.3 is accurate and the error metrics between the computed and the results at the output ports are very low, the hierarchical-based estimations do not consider accurate or realistic values from flattened synthesis. To tackle this, Fake Timer corrects the estimated delays in Section 8.2 to minimize the difference between the computed ATs in Section 8.3 and the results after flattened synthesis. Using corrected delays and ATs, RATs and slacks are recomputed. This strategy ensures that realistic timing is back-annotated to RTL blocks, even if their boundaries do not appear in the synthesized netlist.

Hence, the last stage of the Fake Timer is optional and minimizes the ATs so that the results at the endpoints correlate best with the flattened ATs. This minimization can be formulated as a global constrained optimization. Such optimization aims to optimize an objective function $f(x)$ with respect to a set of variables x, subject to upper b_u and lower b_l bounds as described in Equation 8.4 [161].

$$\min_x f(x), \quad \text{subject to } b_l \leq x \leq b_u, \tag{8.4}$$

where the variable x can be a float or integer. Additionally, the function $f(x)$ can exhibit either convex or nonconvex behavior. This implies it may possess a single global minimum or multiple local minima, which may not necessarily correspond to a global minimum. The objective function delineates a cost function to be minimized or a reward function to be maximized. This objective function can exhibit diverse characteristics; it might be a known, differentiable function, or it could serve as a *black box* or *oracle*. Despite its form, the fundamental goal remains to optimize this objective function.

To estimate flattened realistic results without compromising RTL block boundaries, this thesis aims to minimize the difference between the computed ATs and the ATs of the flattened gate-level netlist. Our proposed approach involves *faking* or correcting the ML-predicted delays to ensure that the *computed* ATs from Algorithm 3 align with the results obtained after flattened synthesis. This process of delay correction is framed as a constrained optimization task,

Algorithm 5 Calculation of boundaries b for AT minimization.

1: **Inputs**: PO: list of primary outputs, $\mathbf{d}$: ML-predicted delays, $\mathbf{AT}$: calculated using $\mathbf{d}$, $\mathbf{AT}^{\mathrm{FLAT}}$: derived from flattened netlist
2: **Outputs**: b: boundaries for all edge delays $\mathbf{d}$ out_diff, L: empty lists ▷ Initialize variables
3: **for** po in PO **do** ▷ Collect differences at POs
4: $out\_diff(po) \leftarrow \mathbf{AT}^{\mathrm{FLAT}}(po) - \mathbf{AT}(po)$
5: **end for**
6: **for** e in $|\mathbf{d}|$ **do** ▷ Back-propagate difference from all POs to all edge delays
7: **for each** po in PO **do**
8: **if** path between e and po **then**
9: Append $out\_diff(po)$ to L
10: **end if**
11: **end for each**
12: $b(e) \leftarrow \mathbf{d}(e) + max(L)$ ▷ Add the maximum difference to the current delay
13: **end for**

ultimately transforming Equation 8.4 into the objective function defined in Equation 8.5.

$$F(x) = \min_{x} \left[\frac{1}{|\mathbf{d}|} \sum_{i=1}^{|\mathbf{d}|} (\mathbf{d}(i) - \mathbf{x}(i))^2 + \frac{1}{|\mathbf{po}|} \sum_{j=1}^{|\mathbf{po}|} |\mathbf{AT}^{\mathrm{FLAT}}(j) - \mathbf{AT}^{\mathrm{CORR}}(j, \mathbf{x}(i))| \right] \quad (8.5)$$
$$\text{subject to} - b(i) \leq \mathbf{x}(i) \leq b(i),$$

where the vector of decision variables $\mathbf{x}$ represents the delays to be corrected. $|\mathbf{d}|$ and $|\mathbf{po}|$ denote the number of delays in $\mathbf{d}$ and primary output ports, respectively. The vector of *faked* or corrected delays x contains floating-point values, and its size depends on each evaluation circuit. Equation 8.5 comprises two terms. The first term corresponds to the MSE between the ML-predicted delays $\mathbf{d}$ and the corrected values $\mathbf{x}$. This term aims at aligning $\mathbf{x}$ with minimal deviation from $\mathbf{d}$. The second term represents the mean absolute difference between the ATs at the output ports, derived from the flattened netlist, and the recomputed ATs, obtained using the new vector of delays $\mathbf{x}$. The general goal is to adjust the delay vector $\mathbf{d}$ to a new value, $\mathbf{x}$, to minimize the difference between the recomputed ATs and the ATs after flattened synthesis. These results are obtained for each primary output and denoted as $\mathbf{AT}^{\mathrm{FLAT}}$. Additionally, the term $\mathbf{AT}^{\mathrm{CORR}}(j, \mathbf{x}(i))$ represents the recomputation of the ATs at the output j using the *faked* vector of delays x.

Lastly, the boundaries b determine the maximum correction that can be applied to each delay $\mathbf{d}$. These are calculated according to Algorithm 5. The highest correction for edges leading to a single output port is determined by the difference between the $\mathbf{AT}$ and $\mathbf{AT}^{\mathrm{FLAT}}$ at that output, as calculated in Lines 6-8 of Algorithm 5. If an edge has paths to multiple outputs, the boundaries must consider the maximum difference in all output ports. The cost function $f(x)$, as defined in Equation 8.5, involves the calculation of the ATs based on the corrected delays x. As the AT computation is a sequence of maximum operators, obtaining its derivative can be computationally expensive. Furthermore, the complexity of this process increases proportionally with the number of delays requiring correction. For this reason, this thesis adopts global heuristic optimization methods to find the vector of corrected delays x that minimizes Equation 8.5. Among the various optimization techniques available, a random search emerges as the most straightforward approach for addressing this problem.

Figure 8.12 depicts the general flow for solving this minimization task. Initially, the two initial steps of the Fake Timer are conducted to obtain the vector of predicted delays x_0 and

the ATs. In particular, primary outputs are annotated with computed ATs. Subsequently, flattened synthesis and timing analysis are performed. From the generated timing reports, the maximum AT values at the output ports are extracted and utilized as a reference for the minimization process. Equation 8.5 is employed to calculate the initial cost value F^0_{min}. The objective is to modify x so that the cost is reduced below F^0_{min}. Before initiating this search, the boundaries, in within the vector $\mathbf{x}$ can be modified, are found as specified in Algorithm 5. With a predetermined number of maximum iterations, different minimization algorithms could potentially be used to modify the initial vector x, such as a random search or through mutations and combinations, as in genetic algorithms. Given the new delay vector $\mathbf{x}'$, the corrected ATs at the POs is recalculated as well as the new cost value. If the resulting cost is lower than F^0_{min}, the current cost value and delay vector are saved as the best values. This iterative process continues until the maximum number of iterations is reached. As previously stated,

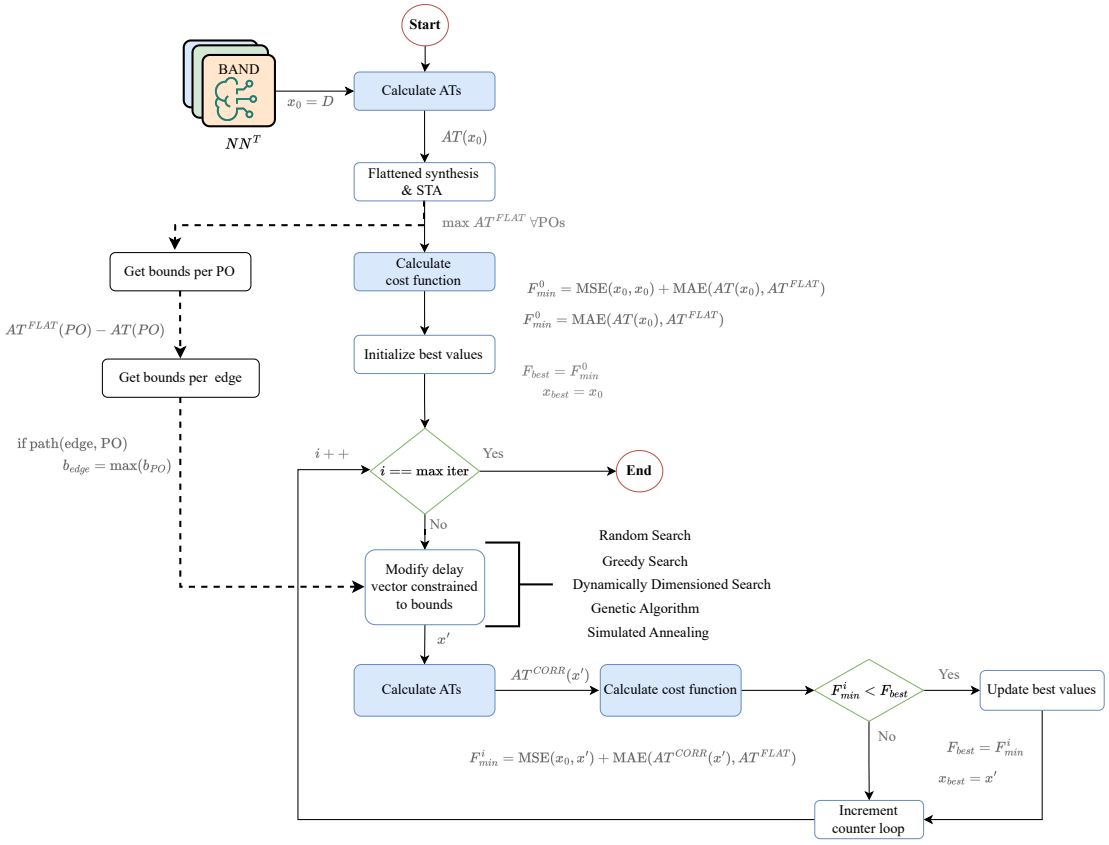


Figure 8.12.: General workflow of minimization task performed by the Fake Timer to align with flattened synthesis results.

Figure 8.12 indicates that the vector $\mathbf{x}$ can be adjusted using diverse strategies. This thesis explores basic heuristic algorithms such as Greedy Stochastic (GS), Dynamically Dimensioned Search (DDS) [162], GAs [163], Simulated Annealing (SA) [164], and a random search, where all elements in $\mathbf{x}$ are randomly modified.

8.4.1. Greedy Search

The greedy search algorithm takes the best immediate local solution or local optimum, which yields globally optimal solutions in a considerable amount of time. The classic GS algorithm is implemented so that for a maximum number of iterations, a random element of the vector $\mathbf{x}$ is modified using a uniform distribution constrained by the boundaries of that given element. Subsequently, the cost function is evaluated for the new vector $\mathbf{x}'$, and if the new cost is lower than the current cost, the best $\mathbf{x}$ vector is updated. This process is repeated for a maximum number of iterations.

8.4.2. Dynamically Dimensioned Search

The DDS algorithm is a global optimization method to solve high-dimensional problems with costly objective functions [162]. Notably, DDS exhibits dynamic behavior, as the number of elements in $\mathbf{x}$ being perturbed from their optimal values decreases as the number of iterations i progresses. The probability that an element is chosen to be perturbed is given by $p(i) = 1 - ln(i)/ln(m)$, where m is the maximum number of iterations. In this way, the search is gradually adjusted from global to local. The selected elements in $\mathbf{x}$ are then modified by a magnitude randomly selected from a normal distribution with a zero mean. The new vector $\mathbf{x}$ is used to compute the cost function, and the vector resulting in a lower cost function is kept as the best solution.

8.4.3. Genetic Algorithm

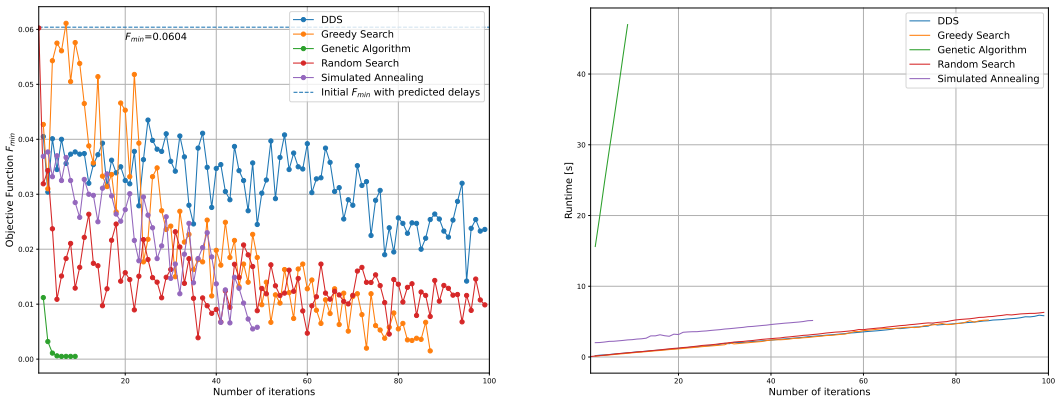
GAs are optimization algorithms inspired by biological evolution, employing mutation, crossover, and selection operators [163]. In each iteration, a *generation* is defined, and its fitness or cost function is computed. Generations with lower cost functions are more likely to be selected to create the next generation. To build a new generation, these candidates are recombined through crossover or modified through mutation. This iterative process is repeated until a predetermined maximum number of iterations is reached. The implementation provided in [165] is used with a probability of 0.8 for mutation and 0.7 for crossover.

8.4.4. Simulated Annealing

This technique approximates the global optimum of a function, mainly when the search space is extensive. It achieves this by employing a temperature parameter, denoted as T, and a rate of cooling, α, which ranges between zero and one. SA selects a random vector $\mathbf{x}'$ close to the current one and compares both based on their cost functions. The new vector is retained as the optimal vector in two scenarios: if its cost is lower than the current vector's cost or if a certain probability p is met. This probability depends on T and is calculated as $p = \alpha T$. Over time, T decreases, which in turn reduces the probability p of accepting worse solutions [164]. The implementation in [166], where 30 initial runs are employed to approximate the optimal values for α and T.

Running Example:

Figure 8.13 presents a comparison of the performance of the mentioned heuristic algorithms when applied to the c17 design. For this design, Fake Timer corrects 24 estimated maximum delays to reduce the error between the computed ATs and the results after flattened synthesis. The initial cost value is $F^0_{min} = 0.0604$, which is exclusively attributed to the second term of Equation 8.5. After exceeding 80 iterations, the best algorithm is the GS, achieving the best cost value of F_{best} =0.015, as illustrated in Figure 8.13a. Even though techniques such as GA and SA show low-cost values, they require longer runtimes, as depicted in Figure 8.13b. These higher runtimes would make this approach impractical for larger designs. Consequently, given its efficiency and performance, the GS technique is selected as the standard minimization algorithm in the Fake Timer flow. GS finds better corrected values of delays and therefore, enables a better localization of critical areas in the designs.



(a) Cost value versus number of iterations

(b) Runtime versus number of iterations

Figure 8.13.: Performance of the minimization algorithms over the c17 benchmark.

8.5. Recomputation of RATs and Slacks

As explained in the previous section, the greedy optimization technique finds a vector of delays that minimizes the difference between the computed ATs in Section 8.3 and the obtained ATs after flattened synthesis. Using this new vector of delays, corrected ATs are recomputed. Afterward, Algorithm 4 can be reapplied to recalculate the RATs for all nodes. Subsequently, slacks can be recomputed using Equation 8.3. This entire process can be done incrementally and results in all nodes being marked with realistic timing metrics that closely resemble the outcomes obtained from flattened synthesis.

Running Example:

Figure 8.14 displays the corrected values annotated to the second row of each pin connection. Across both primary outputs, the ME is 2 ps, and the MAPE is 0.06%. As all slacks are

positive, no critical blocks are found in the RTL IR. Nevertheless, the presence of very high slack values suggests that there may be opportunities for area optimizations.

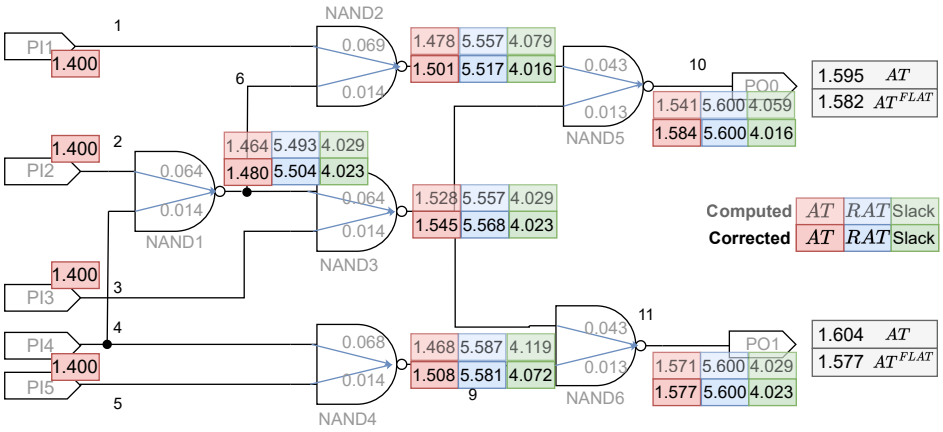


Figure 8.14.: Computed and corrected ATs, RATs, and slacks obtained by the Fake Timer flow.

8.6. Summary

Chapter 6 and Chapter 7 culminate in the presentation of our consolidated estimation engine, Fake Timer. This engine incorporates all the experiments and discoveries discussed throughout the preceding chapters. The Fake Timer flow comprises three significant steps: first, the estimation of pin-to-pin delay and slew, followed by the implementation of a block-based STA flow utilizing the predicted values, and finally, the minimization of the Arrival Times (AT) to closely align with realistic results post-flattened synthesis. In the first step, tabular datasets are collected, and ML models are trained for each component type. Subsequently, a traditional block-based STA approach is executed, involving a forward pass to propagate maximum delays and slews and to compute ATs as well as a backward pass to propagate the RATs and to compute slacks. Especially this block-based method offers two key benefits: it prevents complexity from path-based approaches that scale exponentially with the number of components, and it delivers valuable timing metrics per pin without restricting them to the design's output ports. The third and last step in the Fake Timer flow aims at aligning the computed metrics with the results after flattened synthesis. This adjustment better mirrors the optimizations conducted by the logic synthesis tools, offering a more realistic identification of critical areas.

Fake Timer is a major contribution of this thesis to the state of the art. This novel estimation engine annotates useful timing metrics to each pin of an RTL IR, emulating results after hierarchical or flattened synthesis without the need to execute any tool for RTL generation, synthesis, or STA. Fake Timer draws inspiration from the works of Ye et al. [135] and Guo et al. [136]. Ye et al. [135] and Guo et al. [136] also estimate delay, slews, and ATs, but they follow different approaches. Ye et al. [135] estimate delays and slews and calculate the ATs by summing the predicted gate delays per path. In contrast, Fake Timer aggregates these values but adheres to the block-based STA approach. Meanwhile, Guo et al. [136] employ ML models to predict ATs and slacks directly. This implies the creation of extra training data for AT

prediction and limits the solution space to the training data distribution. Furthermore, they utilize pre-routing netlists as inputs and enable design exploration but at later stages of the design flow. In contrast, Fake Timer takes the RTL IR as input, tackling a more challenging task across various levels of abstraction. Ultimately, Fake Timer enables the identification of critical pins in any design and paves the way for design optimizations at the initial stages of the design flow.

9. Evaluation on Real Designs

This chapter consolidates the findings from the pin-to-pin delay and slew estimation in Chapter 6, the estimation of the maximum ATs using GNNs in Chapter 7, and the computation and correction of the ATs using the Fake Timer estimation engine from Chapter 8. The main focus lies in presenting the outcomes of each method across a set of evaluation designs, which are notably more complex and extensive than the training datasets.

A common element across all three contribution chapters is the data generation workflow, extensively outlined in Chapter 5. Using this automated flow, evaluation designs are easily generated as well. The methods proposed in this thesis are evaluated over different circuits, from hardware adders with different microarchitectures and bit widths to other industrial designs, such as multipliers and encoders. Additionally, circuits of the open-source ISCAS'85 benchmark [158] are employed. Each chapter utilizes distinct data structures, features, and ML model architectures. Hence, details of the evaluation circuits and experimental setups are given separately. In the following, the results obtained for each chapter are summarized.

9.1. Delay and Slew Estimation

In Chapter 6, different training datasets are generated to explore distinct features and ML techniques. The main objective is to train ML models to approximate the delay and slew of pin-to-pin connections within primitive components in an RTL IR. ML models are trained to approximate the delay and slew of pin-to-pin connections within primitive components in an RTL IR. Even though results over the test dataset are higher than 86%, actual designs are far more complex and comprise more components and pin-to-pin connections. Thus, this section presents the inference results of those models over more complex designs. This is achieved by employing evaluation metrics defined in Section 3.1.6. An automated data generation flow allows to quickly generate RTL IRs and the feature structure (whether tabular or graph-based) required by the target ML model. This streamlined approach ensures efficiency in generating the necessary inputs for the models. The pre-trained models are then used to predict slews and delays of pin-to-pin connections within the more complex designs. Subsequently, the estimated values are compared to the pin-to-pin results from timing reports using the same synthesis and STA configuration utilized during the training phase.

9.1.1. Experimental Setup

We employ unseen designs, such as hardware adders with diverse microarchitectures, to assess the various models trained with the different datasets. Adders are an essential component of digital designs, especially of the Arithmetic Logic Unit (ALU), and they are leveraged to build

more complex IPs, such as multipliers and subtractors. Arithmetic operations are indispensable in real-time systems, digital signal processing, and modern AI accelerators.

The simplest version of an adder is the Ripple Carry Adder (RCA). An N-bit RCA is built using N full adders, where the carry-out bit of the previous stage becomes the carry-in of the current stage. At any particular time, only one full adder instance is used [167]. More complex architectural versions are the *Parallel Prefix* adders, as they split the computation into three stages: Pre-processing, carry-propagation, and post-processing [168]. Three main architectures of parallel prefix adders that optimize further for area or performance are Sklansky Adder (SKA) [169], Brent-Kung Adder (BKA)[170], and Kogge-Stone Adder (KSA) [171].

MetaRTL implements these adder architectures for $4, 8, 16, 32$, and 64-bit operations with one carry input and carry output bit. These adder microarchitectures are constructed based on three fundamental types of primitives: Bitwise XOR, AND, and OR. Furthermore, the maximum number of inputs per gate is two. To compare the RTL implementation details of the three parallel prefix adders, key metrics such as the number of components by type, pin-to-pin connections, and maximum path length are documented for the 4-bit microarchitecture in Table 9.1. Notably, this comparison reveals that the SKA architecture exhibits the largest count of components and connections among the three architectures.

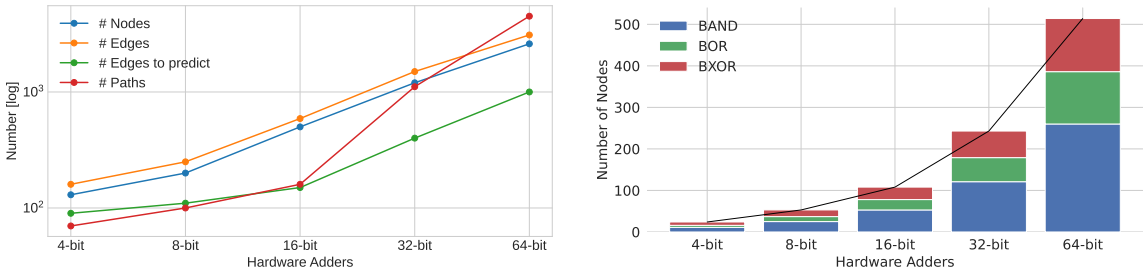
Table 9.1.: Implementation details for the 4-bit adders.

Circuit	No. BXORs	No. BANDs	No. BORs	Pin-to-pin Connections	Max. Distance
BKA	8	7	3	36	7
KSA	8	8	4	40	9
SKA	8	13	8	58	9

To evaluate the GNN-based models, each evaluation design undergoes a mapping process to transform it into a DAG, as depicted in Figure 6.3. Once the designs are converted into DAGs, essential information is gathered to assess their complexity and size. Figure 9.1a presents the averaged values of these measures across all architectures for each bit width. Notably, the figure demonstrates a consistent increase in the number of nodes, edges, edges to predict, and nodes in the longest path as the bit width increases. Similarly, Figure 9.1b illustrates the number of nodes in which the node type feature is a logic gate, such as BAND, BOR, and BXOR. Nodes representing primary ports, pins, or other components are excluded from the figures. Additionally, the most common node degree for all DAGs is two, indicating that most nodes have an average of two incident edges. Finally, except for the RCA and its vectorized implementation RCAV, which exhibit up to six levels of depth, the maximum depth level for all adder graphs is four.

9.1.2. Regression Performance

Chapter 6 encompasses the utilization of a total of three distinct datasets, namely DS-I, DS-II, and DS-III, in conjunction with four different model architectures, including an MLP, TabMLP, GCN, and MPNN. Table 9.2 summarizes the different experiments conducted by combining the



(a) Number of nodes and edges

(b) Number of nodes of logic type

Figure 9.1.: Averaged graph information over all adders for each bit width.

Table 9.2.: Overview of experiments predicting pin-to-pin delays and slews from Chapter 6.

Experiment ID	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
Dataset	*DS-I*				DS-II		DS-III	
Propagated Slew	✓	✗	✓	✗	✓	✓	✓	✓
ML Architecture	MLP		TabMLP		MLP	TabMLP	GCN	MPNN

different datasets with the respective ML model capable of handling the specific data structure. For DS-I, experiments are conducted with and without considering slew as a feature. Similarly, the same model architectures are employed for DS-II with the distinction that slews are always taken into account. Finally, DS-III uses graphs as inputs; thus, only the GNN-based models can be utilized. In the following sections, the evaluation circuits employed are introduced, and the results of each experiment are discussed.

9.1.2.1. Using Dataset I

The experiments in this section are the first four listed in Table 9.2. They focus on evaluating the performance of the MLP and TabMLP models when trained using DS-I. The primary focus lies in examining the implications of simultaneously learning and propagating slew values to the subsequent connected components within the timing path. Moreover, only 4-bit hardware adders are used for evaluation.

Table 9.3 provides an overview of the averaged R^2 value for the delay estimation, revealing that the MLP and TabMLP models, without slew information in the training features, exhibit a notably lower estimation accuracy. On the other hand, including slew values in the dataset yields a marked improvement in accuracy, aligning with the intuitive understanding that the pin-to-pin delay is intrinsically linked to the transition time of the input signal. This relationship is also explicitly seen in the timing models used in technology libraries. However, based on the STA configuration used, the input slew is known only for the primary ports. Consequently, the previously predicted output slew must be propagated and utilized as the input slew for the following pin connections in the timing graph. This implies that the inference process for a complete circuit should be done sequentially, component by component. Even though sequential inference is slower than parallel inference, where all pin-to-pin connections are considered

Table 9.3.: R^2 of MLP and TabMLP with and without considering slew propagation.

	Slew not considered			**Slew Considered**			**Slew Propagated**		
Test Circuit	BKA	KSA	SKA	BKA	KSA	SKA	BKA	KSA	SKA
MLP R^2	0.31	0.36	0.37	0.95	0.92	0.95	0.94	0.92	0.94
TabMLP R^2	0.31	0.30	0.37	0.84	0.87	0.91	0.84	0.88	0.90

Table 9.4.: Hyperparameters of ML shallow models.

Regressor	**Hyperparameter**	**Value**
Random forest	Number of trees	400
	Minimum sample split	2
	Minimum sample leaf	1
Decision tree	Criterion	Squared error
	Splitter	Best
	Minimum sample split	2
	Minimum sample leaf	1

simultaneously, sequential inference aligns with the natural timing analysis flow. The final row of Table 9.3 affirms this strategy and displays high R^2 values. The same behavior is observed with TabMLP. Based on the results for DS-I, all following experiments always consider learning and propagating the slew values.

The results presented in Table 9.3 are derived under consistent timing constraints, including clock configuration, delay, and slews for primary ports, mirroring the configuration used during the training phase of the ML models. In an effort to assess the models' stability across varying input slew configurations, the inference is run for two known configuration variants and one unseen. On average, The MLP model achieves an MAE of 6.76 ns, an MSE of 0.96 ns^2, and an R^2 coefficient equal to 0.87. The experiments are repeated using the TabMLP model, which averages 9.49 ns MAE, 1.39 ns^2 MSE, and R^2 equal to 0.79. Notably, the performance of the MLP model surpasses that of the TabMLP model in this scenario. However, training ML models with dataset DS-I imposes limitations on the supported component types, confining it solely to logic operators. Considering that real-world designs encompass a broader spectrum of component types beyond logic operators, the need for adaptable models capable of accommodating this diversity becomes increasingly apparent.

9.1.2.2. Using Dataset II

Based on the results for DS-I, the consideration of learning and propagating slew values is extended to the experiments conducted with DS-II, as delineated in Table 9.2. In contrast to DS-I, DS-II considers more RTL IR primitives beyond logic operators, i.e., one single model is trained to predict the timing behavior of pins within different component types in a design, such as multiplexers, comparators, logic and arithmetic operators. To evaluate the stability of the models across different input slew configurations, the inference process is executed for three distinct configurations. Furthermore, shallow methods, such as RFs and Decision Trees (DTs), are configured with the hyperparameters detailed in Table 9.4 and trained using DS-

Table 9.5.: Evaluation of models trained with DS-II for adders with different input slews.

Test Circuit	R^2				MAE [ns]				MSE [ns^2]			
	MLP	**TabMLP**	**DT**	**RF**	**MLP**	**TabMLP**	**DT**	**RF**	**MLP**	**TabMLP**	**DT**	**RF**
BKA	0.66	0.75	0.51	0.55	8.06	7.30	8.98	8.76	0.98	0.73	1.46	1.23
KSA	0.68	0.77	0.52	0.53	7.70	7.11	8.54	8.81	0.97	0.72	1.40	1.24
SKA	0.68	0.81	0.53	0.53	8.03	6.32	9.34	9.71	1.01	1.71	1.66	1.46
Average	0.67	**0.77**	0.52	0.54	7.94	**6.91**	8.96	9.09	0.99	**0.68**	1.51	1.31

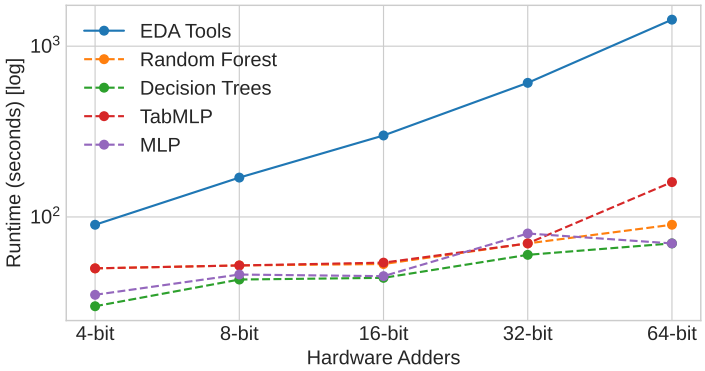


Figure 9.2.: Runtime comparison between ML models and EDA tools.

II. As explained in Chapter 3, these shallow methods diverge from deep NNs by focusing on decomposing the data into sets of criteria rather than constructing stacked layers of neurons. A primary advantage of these shallow methods over NNs lies in their interpretability, offering a more transparent approach to data analysis and modeling.

Table 9.5 presents the prediction results from the inference process conducted over the 4-bit adders, utilizing both shallow and deep ML models. Notably, this table shows that the TabMLP generalizes slightly better to more RTL primitives than the MLP model. This can be attributed to the adaptability of the first embedding layer in the TabMLP model, which effectively adapts to a higher number of categories or component types when compared to the sparse vector obtained through the one-hot encoding format in the MLP architecture. The deep models generally outperform both shallow regressors in all evaluation metrics and circuits. Interestingly, despite the disparities in model performance, the runtimes of the four models remain comparable, as depicted in Figure 9.2. This observation highlights the main advantage of employing ML models for design metric estimation, as these models exhibit an average inference runtime that is 8.4x faster than running the EDA tools. This improvement might seem modest at first glance; however, as illustrated in Figure 9.2, the speed-up scales exponentially with the design size. In addition to reducing runtime, this approach also minimizes the time and effort required by the engineer to configure logic synthesis and STA tools. Consequently, faster timing estimations are achieved through the one-time effort of collecting datasets and training ML models.

Reviewing the insights from Table 9.5, it becomes evident that the TabMLP architecture

outperforms the MLP model. TabMLP achieves an average MAE of 6.91 ns, an MSE of 0.68 ns^2, and an R^2 value of 0.77, while the MLP model yields average MAE, MSE, and R^2 values of 7.94 ns, 0.99 ns^2, and 0.67, respectively. However, the quality of the predictions for the DS-II decreases compared to the results for DS-I, a phenomenon attributed to three key factors. First, as the number of supported component types increases, the feature matrix becomes sparser, potentially impacting the model's learning process. Additionally, the range of RTL features and their contribution to the prediction depends on the specific component type. For instance, a MUX has multiple inputs, while a NOT has only one. For the latter, the number of inputs is irrelevant and may disturb the learning process. Third, the component type also impacts the slew and delay ranges, suggesting that a global normalization or scaling may destroy some patterns of the objectives. For instance, logic components at RTL are mapped to logic cells of the technology, while arithmetic components are transformed into a set of logic gates implementing that functionality. Thus, the delay and slew values are lower for logic components than for arithmetic operators at the RTL.

9.1.2.3. Using Dataset III

This section details the results obtained for the last two experiments in Table 9.2. These experiments exclusively employ the end-to-end GNN models, specifically the MPNN model and the GCN-based model, to predict the slews and delays between input and output pins of the same component. In essence, the estimation process focuses solely on the edges highlighted in blue in Figure 6.3. It is important to note that values for other edges, such as those connecting primary ports to pins or the output and the input pins of different components, are treated as ideal wires and are not considered for prediction.

Inference of all models is executed for all available implemented adder microarchitectures and bit widths. The resulting R^2 coefficient obtained through the GCN-based architecture is visually presented in Figure 9.3. Notably, the average R^2 coefficient across all adders and bit widths stands in average at 0.78, with the best results observed for 4-bit (around 0.96) and 8-bit (around 0.92) bit widths across all adder microarchitectures. However, adders with higher bit widths, such as 16, 32, and 64-bit, include pins, with a bit width larger than 8-bit setting R^2 partially below 0.5. Reason is that these cases of wide bits have not yet been covered in the training data. Additionally, the R^2 scores when using the MPNN are worse than in Figure 9.3. This observation demonstrates that despite incorporating multidimensional edge features for node embedding updates, the MPNN performs inadequately on larger or unseen graphs. This finding confirms the known weakness of MPNNs in distinguishing between two isomorphic graphs, where topologically identical graphs are embedded to the same point [172]. Contrarily, the GCN-based model outperforms the MPNN, demonstrating an averaged MSE and MAE of 0.16 ns^2 and 2.76 ns, respectively.

9.1.3. Summary and Conclusions

Table 9.6 summarizes the experiment results, revealing the high complexity inherent in the pin-to-pin delay estimation task. According to the results, the GCN-based model outperforms

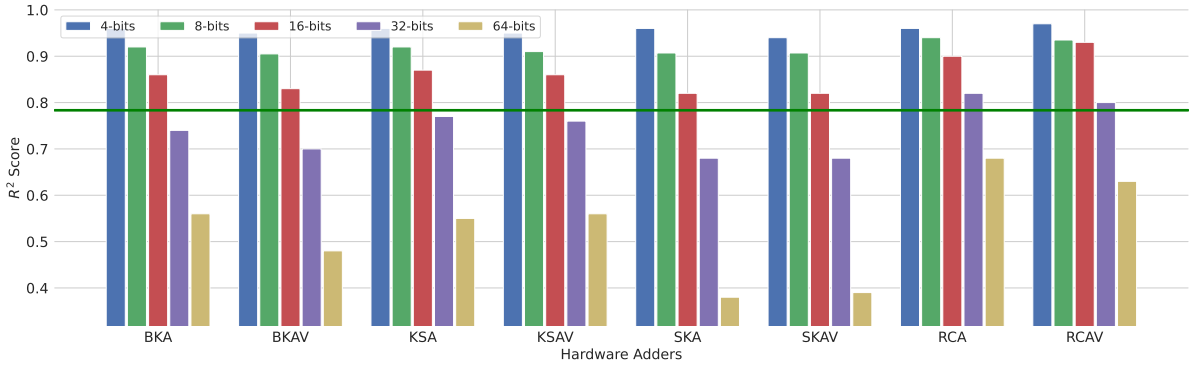


Figure 9.3.: Average R^2 for DS-III using the GCN-based model.

all other methods in terms of performance across 4-bit and tested adders. Moreover, taking inspiration from technology libraries proves to be beneficial in effectively representing the nature of the task. This becomes particularly evident with the increase in model accuracy after introducing slew as input and objective features. However, despite this improvement, the best R^2 coefficient remains 0.78, and we must note that the GCN model is trained for DS-III, containing solely logic components. Given that real-world designs encompass multiple component types, it becomes essential to adopt a more generalizable approach. Each component type exposes unique functionalities, pin configurations, bit widths, and time behaviors. Consequently, the inclusion of all component types within a single model for training introduces noise to the training data, potentially compromising the quality of predictions.

Table 9.6.: Performance of models when trained with different datasets and evaluated over hardware adders.

Test Dataset	**Training Dataset**	**Model**	R^2	**MAE**	**MSE**
4-bit	DS-I	MLP	0.87	6.76	0.96
		TabMLP	0.79	9.49	1.39
	DS-II	MLP	0.67	7.94	0.99
		TabMLP	0.77	6.91	0.68
	DS-III	MPNN	-0.83	20.67	7.04
		GCN	**0.96**	**2.64**	**0.16**
All bit widths	DS-II	MLP	0.59	7.00	1.00
		TabMLP	0.73	7.00	1.00
	DS-III	MPNN	-1.00	22.89	7.81
		GCN	**0.78**	**0.16**	**2.76**

The inference flow of ML models offers a notable advantage in terms of speed compared to the conventional flow involving EDA tools. Compared to non-ML-based estimation approaches, the process of training, validating, and evaluating an ML model over training data is generally more accessible than formulating analytical, macro modeling, and polynomial models. However, the ML model's performance is significantly impacted by several factors, such as the selected training circuits, the covered component types, and the primary port's timing constraints. Moreover, ensuring similarity in the data distribution between the training and

test sets, particularly for timing-related features and objectives, presents a notable challenge. Addressing this through the consideration of just a couple of timing constraints for primary ports may not suffice.

In conclusion, the models and experiments in this section prove the complexity of the delay estimation task and motivate the steps conducted in Chapters 7 and 8. These approaches involve implementing more strict configurations for training circuits to cover necessary configurations rather than all possible ones, as well as training an ML model for each component type to focus on the features and timing behavior of one component type at a time. Furthermore, the inclusion of more diverse timing constraints for the training circuits, particularly concerning the delay and input slews of primary ports, is emphasized. Lastly, it is crucial to recognize that the criticality of individual pins within a component may not provide sufficient information about the critical paths of the entire design. Therefore, it is essential to extend the pin-to-pin delay estimation to enable the comparison of RTL variants and the precise identification of critical regions within the design.

9.2. Arrival Timing Estimation using GNNs

Chapter 7 trains different GNN models to approximate the delay and slew of pin-to-pin connections within primitive components in an RTL IR. The latter is similar to the flow in Chapter 6. However, this chapter introduces a new dimension by concurrently estimating the maximum or worst arrival times across all primary outputs. Even though results over the test dataset are higher than 97%, actual designs are more complex, characterized by graphs with a higher number of nodes, edges, and depth levels. Thus, this section presents the inference results of those models over larger designs specifically summarizing the outcomes of the joint estimation of pin-to-pin delays and slews, alongside the maximum AT of a design across all its endpoints.

The training flow in Figure 7.2 is simplified, taking out steps necessary for ground-truth label collection. Figure 9.4 shows the only required steps for running inference over the two GNN-based models. The automated data generation and graph collection flow allows to quickly generate RTL IRs and the featured DAGs. In contrast to the training flow, the generation of the RTL, gate-level netlists, and timing reports are skipped, as featured graphs are built directly from the MoDs or RTL IRs. However, for comparison purposes, the RTL design generation, synthesis, and STA runs are conducted to collect pin-to-pin results and worst arrival times.

9.2.1. Experiment Setup

To evaluate both GNN-based architectures, a total of 50 Hardware (HW) adders, directly generated using MetaRTL, are utilized along with six circuits taken from the open-source ISCAS'85 benchmarks [158]. The hardware adders are already introduced in Section 9.1.1. The evaluation designs use different microarchitectures of adders from RCA to parallel prefix adders and have different input bit widths and vectorized implementations.

The ISCAS'85 [158] benchmarks are a set of ten combinatorial gate granular digital designs.

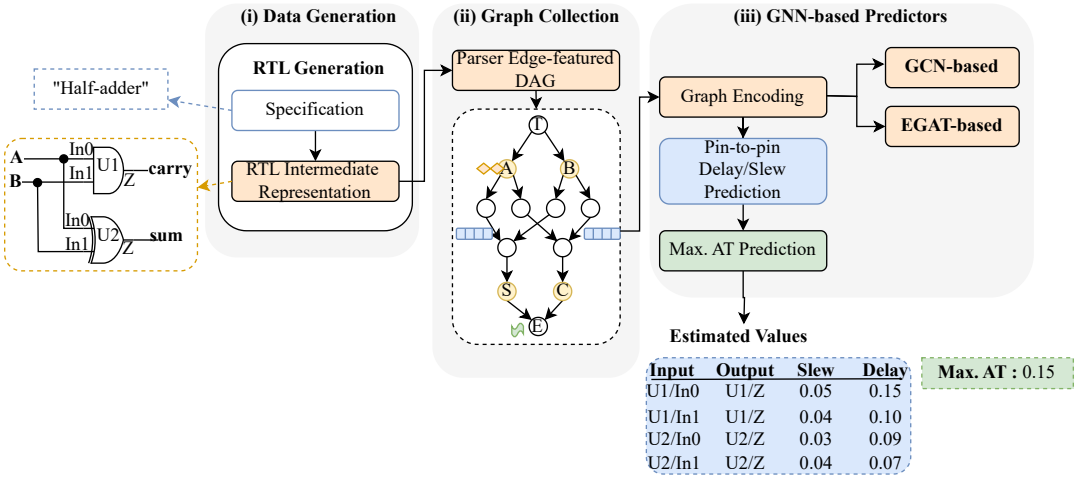


Figure 9.4.: Flow for prediction of delays, slews, and ATs from RTL IR designs.

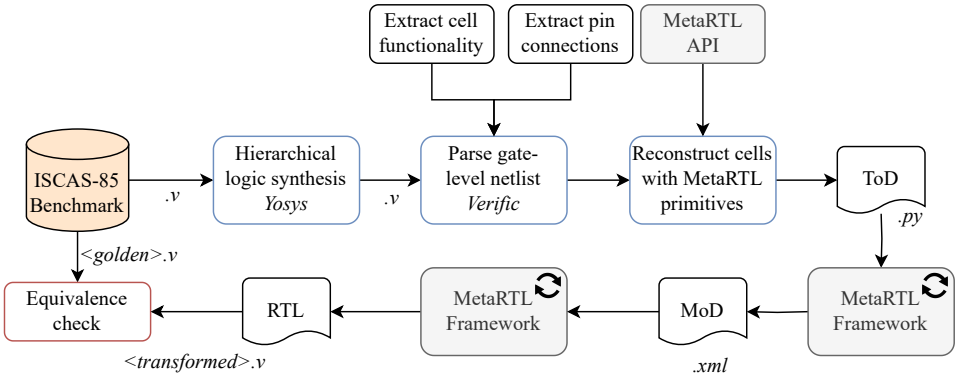


Figure 9.5.: Flow overview to import ISCAS'85 designs in Verilog to MetaRTL.

These designs cover different IPs, such interrupt controllers, error-correcting units, and ALU implementations. Specifically, this chapter employs the *c17*, *c432*, *c499*, *c1908*, *c1908a*, and *c1355* designs. The ISCAS-85 benchmarks are provided in Verilog code. To import those benchmarks into the MetaRTL framework, the flow depicted in Figure 9.5 and introduced by Kaja et al. [173] is followed. First, the Verilog code is hierarchically synthesized using Yosys. The synthesized netlist is parsed using Verific [174]. Given the function expressions of each gate in the technology library, the netlist's functionality is reconstructed using components supported in the RTL generation framework to obtain their RTL intermediate representations. The reconstructed design has then a mixed register-transfer and gate-level granularity. Finally, a new version of the RTL is generated, and equivalence checks are run to verify that the original and the reconstructed Verilog code have identical functionality. Using the generated MoD, the featured graphs are constructed and can be employed for inference.

Table 9.7 lists the relevant features of some evaluation designs to give an idea about their complexity. In total, 56 evaluation circuits are employed. Each design is synthesized by performing 20 different STA runs for each design. Thus, 1120 different featured graphs are

Table 9.7.: Evaluation benchmarks for Chapter 7.

50 Hardware Adders	**# Primary inputs**	**# Primary outputs**	**# Nodes**	**# Edges**	**Depth**
Brent-Kung 4-bit	2	2	124	148	20
Kogge-Stone 8-bit	2	2	227	272	36
Sklansky 16-bit	2	2	704	881	72
...	...	...	...	...	...
Ripple Carry 32-bit	2	2	1221	1427	296
Vect. Brent-Kung 64-bit	2	2	2851	3495	54
6 ISCAS'85	**# Primary inputs**	**# Primary outputs**	**# Nodes**	**# Edges**	**Depth**
c17	5	2	52	58	18
c432	36	7	1294	1459	106
...	...	...	...	...	...
c1355	41	32	1586	1802	66
c1908	33	25	2246	2563	92

generated.

9.2.2. Regression performance

Chapter 7 explains how the two GNN-based models jointly solve two regression tasks. The first task predicts the pin-to-pin delay and slew values similar to Chapter 6. The second task predicts the maximum AT of a design. Therefore, the performance of the models is evaluated regarding both tasks. To that end, evaluation metrics introduced in Section 3.1.6 are employed, such as the coefficient of determination R^2, MSE, MAE, and ME.

9.2.2.1. Prediction of Pin-to-Pin Delays and Slews

The performance of the models in the edge-level prediction task is described in Table 9.8, where the actual values are acquired directly from the STA reports. The average results demonstrate that the GCN model outperforms the EGAT model in the slew prediction, achieving an R^2 score of 0.89 and a ME of 0.12 ns. However, the EGAT model excels in the delay prediction with a ME of 0.58 ns. Both models predict the output slews more accurately by considering the input slews as one of the edge features. The GCN effectively captures underlying patterns by propagating information equally from all neighbors, while the EGAT, equipped with a multi-head attention mechanism and 24K parameters, struggles to generalize to unseen designs. It is important to note that both models yield bad or negative average R^2 scores for the delay prediction. Further analysis reveals that this decrease in R^2 scores relates to the design complexity. For instance, the average R^2 for delay estimation in 4-bit adders reaches 0.62, whereas it drops to 0.16 for 16-bit adders. This can be tackled by making the model deeper and increasing model complexity, which may lead to over-smoothing issues and longer training times. Thus, the models represent a good trade-off between model complexity, training time, and performance.

Table 9.8.: Results for the maximum delay, slew, and arrival time predictions.

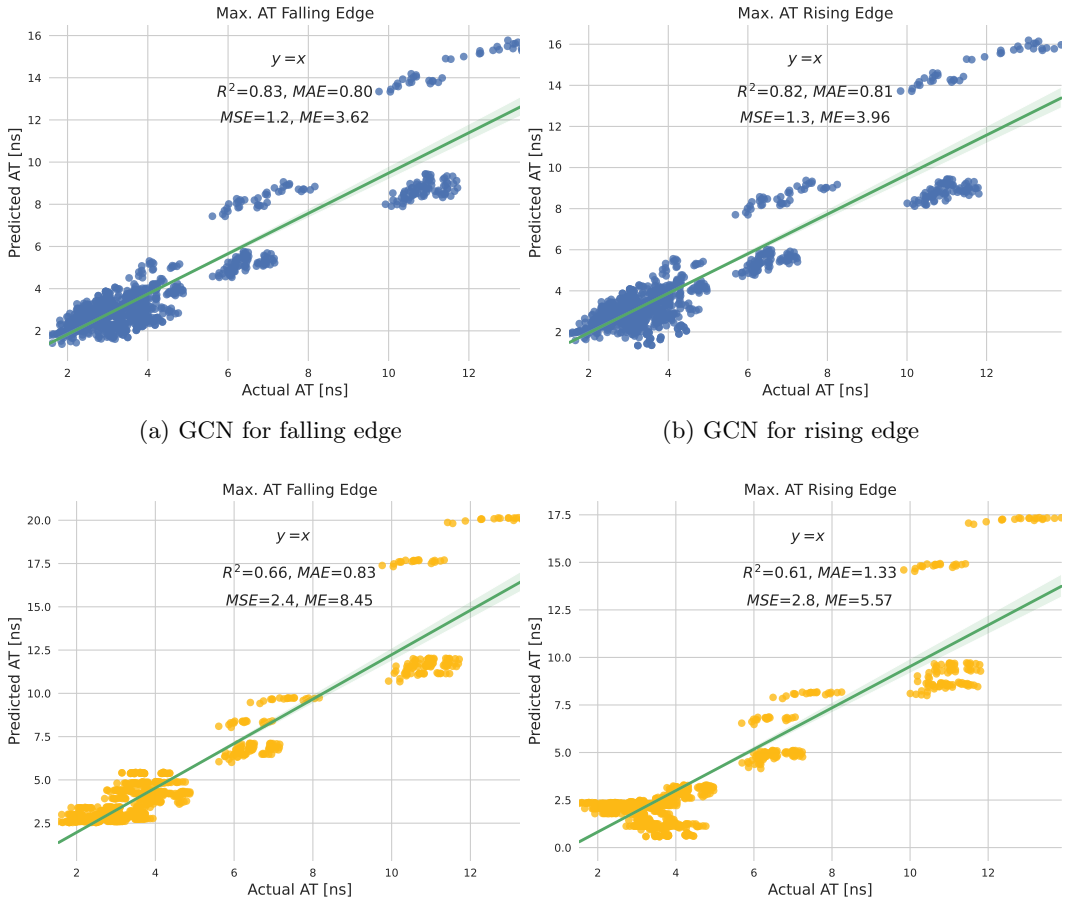
Objective	GNN-based Models							
	GCN				EGAT			
	R^2	MAE [ns]	MSE [ns^2]	ME [ns]	R^2	MAE [ns]	MSE [ns^2]	ME [ns]
Slew	**0.89**	**0.02**	**0.00**	**0.12**	0.58	0.04	0.01	0.38
Delay	-0.98	0.19	0.08	0.74	**-0.06**	**0.16**	**0.04**	**0.58**
Max. Arrival Time (AT)	**0.82**	**0.81**	**1.3**	**3.79**	0.64	1.08	2.6	7.01

9.2.2.2. Prediction of Maximum ATs

Figure 9.6 depicts the regression results for rising and falling edges using the GCN (top row) and EGAT models (bottom row). The continuous line within each plot denotes the ideal scenario where the prediction aligns perfectly with the ground-truth values. Each data point in a plot corresponds to an evaluation graph corresponding to a hardware adder or the ISCAS'85 benchmarks with one of the 20 random STA configurations. Furthermore, Table 9.8 lists the average regression metrics for the AT prediction task in the last row. Averaging for both timing edges in Figure 9.6a and 9.6b, the GCN model achieves an R^2 of 0.82, with an MAE of 0.81 ns, an MSE of 1.3, and an ME of 3.79 ns. Notably, the GCN model outperforms the EGAT model, as evidenced by the EGAT model's R^2 score of 0.64, obtained when the number of training epochs is doubled. Averaging for both timing edges in Figure 9.6c and 9.6d, the EGAT model demonstrates an MAE of 1.08 ns, an MSE of 2.6, and an ME found of 7.01 ns. These results are particularly promising, considering the input to the GNNs models is an RTL IR, not a detailed gate-level netlist. Our approach's main benefit is obtaining pin-to-pin slews, delays, and AT estimations faster than running RTL generation, synthesis, and STA tools. Figure 9.7 compares the inference time in milliseconds for the models using GCN and EGAT architectures, depicted with a blue and yellow line, respectively, while the tools' runtimes in seconds are represented by a green line. Each evaluation circuit is represented as one point on the x-axis, ordered in increasing order based on the runtime required by the conventional flow using EDA tools. Thus, smaller designs, such as the c17 benchmark, require considerably less time than larger designs, such as the c1908 benchmark. The experiments are conducted on the same Linux machine with an Intel® Xeon(R) Gold 6126 CPU at 2.60 GHz. On average, the runtime of the tools for the 56 evaluation designs is $930{\times}10^3$ ms, while the GCN and EGAT models achieve impressive inference times of 8 ms and 16 ms, respectively. In summary, the inference runtimes of both models are three orders of magnitude faster than running the EDA tools.

9.2.3. Summary and Conclusions

This chapter is the first to employ GNNs to estimate pin-to-pin delay, slews, and maximum ATs from an RTL IR. To that end, the two GNNs that support multidimensional edge features are compared: the extended GCN and EGAT. The resulting embedding vectors from both GNNs serve as inputs to two MLPs. The evaluation of unseen designs demonstrates the superior performance of the GCN model over the EGAT model, achieving an R^2 score of 0.82 and an ME of 3.72 ns for AT predictions, even with fewer training epochs than required to train the



(a) GCN for falling edge

(b) GCN for rising edge

(c) EGAT for falling edge

(d) EGAT for rising edge

Figure 9.6.: Maximum AT regression performance for both timing arcs and models.

EGAT model. Our models represent a good trade-off between complexity and performance, offering a significant advantage with inference times that are three orders of magnitude faster than the combined runtimes for RTL generation, synthesis, and STA.

Furthermore, the models and experiments in this chapter also show how challenging timing estimation tasks are. Even though both models have a high R^2 coefficient on the AT and slew prediction, the results for the delay estimation are inferior to the outcomes obtained in Chapter 6. These findings confirm even more the intuition that a single model and a single training flow covering all possible component types and multiple tasks is an overly complex task for one single model. Finally, while identifying the maximum AT among all design outputs facilitates the comparison of critical ATs among different RTL variants, it does not allow for pinpointing specific components in the design that can be optimized. Thus, the findings in this chapter motivate the block-based STA flow leveraging ML models introduced in Chapter 8.

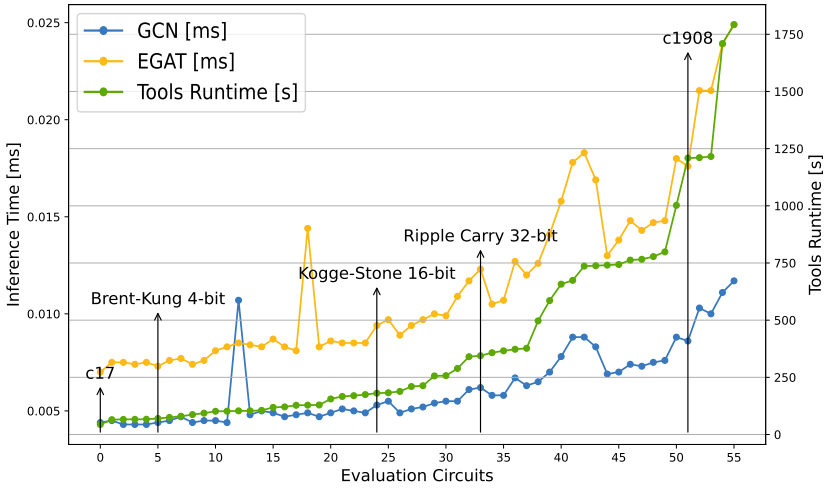


Figure 9.7.: Runtimes of model inference and EDA tools for 20 runs of each evaluation circuit.

9.3. Fake Timer Performance

The Fake Timer flow presented in Chapter 8 builds upon the findings of previous chapters and represents a consolidated timing estimation engine at RTL. Chapter 8 explains the three main steps of the Fake Timer flow, starting with the pin-to-pin delay and slew estimation, followed by block-based calculations using predicted values to compute ATs and finalizing with the minimization of the difference between the computed ATs and the resulting ATs after flattened synthesis. Additionally, Chapter 8 demonstrates the practical applicability of the Fake Timer flow over the simple c17 benchmark. This section presents the results of running Fake Timer over a more extensive evaluation dataset. Specifically, the computed and the corrected ATs are compared with results from the OpenSTA tool when conducting hierarchical and flattened synthesis.

9.3.1. Experiment Setup

The Fake Timer's effectiveness is assessed using 83 designs taken from both the open-source ISCAS'85 [158] benchmarks and industrial datasets. MetaRTL directly generates industrial designs, such as the hardware adders, multipliers, and encoders delineated in earlier sections. The ISCAS'85 [158] benchmarks are integrated into MetaRTL by creating a gate-RTL mixed representation, as depicted in Figure 9.5. This section expands the evaluation dataset discussed in Section 9.2, introducing larger industrial designs, including vectorial and array multipliers and encoders. Moreover, larger ISCAS'85 designs are also considered, such as the *c2670*, *c3540*, and *c5315*, which functions correspond to a 12-bit, 8-bit, and 9-bit ALU, respectively [175]. Table 9.9 lists some of the evaluation designs to provide an understanding of their complexity. This section presents the performance of the Fake Timer over both design sets for the AT computation using ML-predicted delays and the correction considering flattened synthesis results. Lastly, a runtime analysis and comparison with related works are provided to offer a comprehensive assessment of the effectiveness of the presented approach.

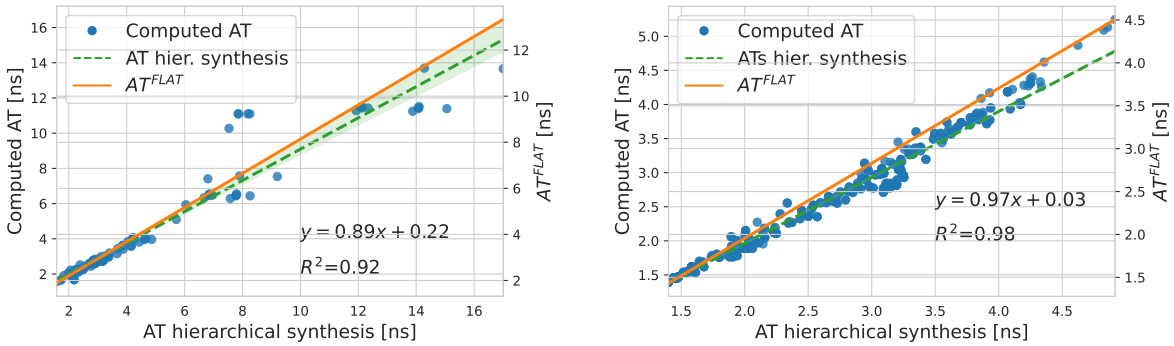
Table 9.9.: Evaluation benchmarks for Chapter 8 (FakeTimer).

72 Industrial Designs	**# Primary inputs**	**# Primary outputs**	**# Pin-to-pin connections**	**# Nodes**	**# Edges**
prio_encoder	1	2	16	94	113
BKA_4bit	2	2	34	124	148
...	...	...	...	...	...
BKA_64bit	2	2	946	2308	2860
SKAV_64bit	2	2	1696	3543	4473
11 ISCAS'85 Designs	**# Primary inputs**	**# Primary outputs**	**# Pin-to-pin connections**	**# Nodes**	**# Edges**
c17	5	2	14	52	58
c432	36	7	521	1294	1459
...	...	...	...	...	...
c3540	50	22	3164	7411	8424
c5315	178	123	4031	9976	11349

9.3.2. Evaluating Computed ATs

The Fake Timer flow calculates ATs, RATs, and slacks for all pins in an RTL IR. It also considers falling and rising edges. For comparison, only the maximum ATs at primary outputs for the rising edge are analyzed. We compare the computed **AT** with the maximum ATs returned by OpenSTA from all paths.

Figure 9.8 shows the computed ATs for both the industrial designs and the ISCAS'85 benchmarks. In the figure, each point marked with blue represents the AT of a primary output in an evaluation circuit. The industrial designs comprise 158 primary outputs, while the ISCAS'85 circuits have 663 primary outputs. The closeness of the points to the dashed line indicates the strength of the correlation between the calculated **AT** and the ATs after hierarchical synthesis. To quantify this correlation, the coefficient of determination R^2 is employed. Remarkably, the average R^2 score stands at 0.95 for both sets of designs. This high correlation can be attributed to training each ML model using timing reports derived from hierarchical netlists.



(a) Industrial Designs (b) ISCAS'85 Designs

Figure 9.8.: Computed ATs using ML predictions versus the ATs derived from hierarchical and flattened synthesis.

Table 9.10 compares the error metrics between the computed ATs and the results following both hierarchical and flattened synthesis. The outcomes show a stronger correlation with the results of hierarchical synthesis. However, as discussed in Chapter 8, the timing analysis of a flattened netlist yields more realistic results, as it considers cross-boundary optimizations. Comparing the computed ATs with the ATs of the flattened netlist, $\mathbf{AT}^{\mathrm{FLAT}}$, the average R^2 score, MAPE, and ME for both design sets are found to be 0.85, 10.26%, and 2.71 ns, respectively. Notably, the ISCAS'85 designs exhibit lower errors than the industrial designs. This can be attributed to the higher number of primary outputs and smaller magnitudes of ATs in the ISCAS'85 designs. Moreover, it is noteworthy that some industrial designs encompass unsupported component types, which are considered ideal wires during the Fake Timer flow, thus impacting the overall error and resulting in outliers in Figure 9.8a.

Table 9.10.: Error metrics for *computed* ATs w.r.t. ATs derived from hierarchical and flattened synthesis.

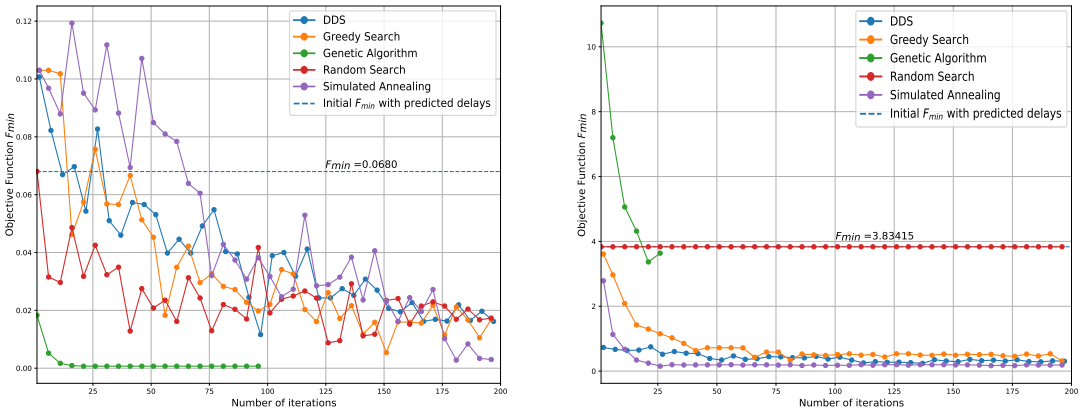
Regression Error Metric	**Industrial Designs**		**ISCAS'85 Design**		**Average**	
	Hierarchical synthesis	**Flattened synthesis**	**Hierarchical synthesis**	**Flattened synthesis**	**Hierarchical synthesis**	**Flattened synthesis**
R^2	0.92	0.77	0.98	0.93	0.95	**0.85**
MAPE [%]	9.56	15.85	2.54	4.67	6.05	**10.26**
ME [ns]	3.66	4.23	0.38	1.19	2.02	**2.71**

9.3.3. Evaluating Corrected ATs

In Figure 9.8, the solid orange line represents the regression line between the computed ATs and the results after flattened synthesis $\mathbf{AT}^{\mathrm{FLAT}}$. The gap between the orange and green lines motivates the minimization proposed in Equation 8.5. This minimization problem is mapped to a constrained optimization task, where $\mathbf{x}$ represents a vector of continuous ML-predicted delays. The size of this vector varies depending on the evaluation circuit. The cost function calculates the ATs in Equation 8.1 for every new vector $\mathbf{x}$. This equation involves a chain of maximum operators, which renders the derivative of the cost function both computationally demanding and non-convex. Fake Timer solves Equation 8.5 using global optimization methods such as SA [164], DDS [162], and GS [176]. These methods are implemented in Python to find the optimal vector of *fake* delays after 50 iterations.

To assess the performance of the different minimization algorithms, 4-bit and 64-bit BKAs are selected as the use cases. The number of delays considering both timing arcs to correct is 68 and 1892 for the 4-bit and 64-bit adder, respectively. Figure 9.9 shows the results of the optimization algorithms for both designs when minimizing the cost function F_{min} in Equation 8.5. For the 4-bit adder, the GA method, outperforms other heuristic methods, reaching a very-low cost F_{min} since the first iterations. However, it performs poorly in higher dimensional problems like the 64-bit adder. Similarly, a random search is inefficient in the 64-bit adder. On the other hand, the DDS, GS, and SA methods successfully found an optimal vector of delays for both use cases. Hence, these methods are identified as the most effective for further analysis.

The three most effective approaches, namely SA, DDS, and GS, are applied over industrial, and



(a) 4-bit BK Adder

(b) 64-bit BK Adder

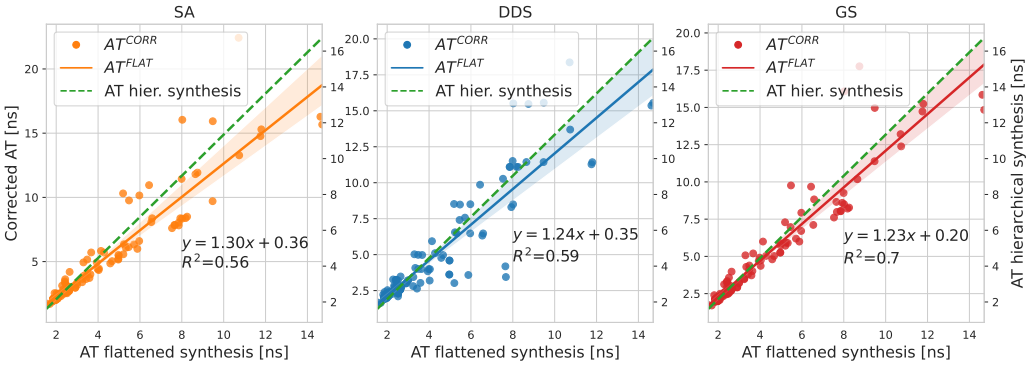
Figure 9.9.: Optimization algorithms over two adder designs when minimizing F_{min} in Equation 8.5 up to 200 iterations.

ISCAS'85 designs to get the vector of fake delays. Leveraging these faked delays, corrected ATs, denoted by $\mathbf{AT}^{\mathrm{CORR}}$, are recomputed. Figure 9.10 illustrates the regression lines for the ATs from hierarchical synthesis (represented by the green dashed line), the flattened ATs (represented by the solid line), and the corrected ATs obtained with each algorithm (depicted with the blue dots). The role of Fake Timer is to fine-tune the delays, aligning $\mathbf{AT}^{\mathrm{CORR}}$ closer to $\mathbf{AT}^{\mathrm{FLAT}}$. Among the ISCAS'85 designs, the DDS algorithm performs the best, achieving an R^2 score of 0.91. On the other hand, GS is the most effective option for the industrial benchmarks, with an R^2 of 0.70. Additional error metrics for the three optimization algorithms are compiled in Table 9.11. For both design sets and the best-performing algorithms, the average R^2 score, MAPE, and ME of the $\mathbf{AT}^{\mathrm{CORR}}$ compared to the $\mathbf{AT}^{\mathrm{FLAT}}$ are found to be 0.81, 10.19%, and 5.10 ns, respectively.

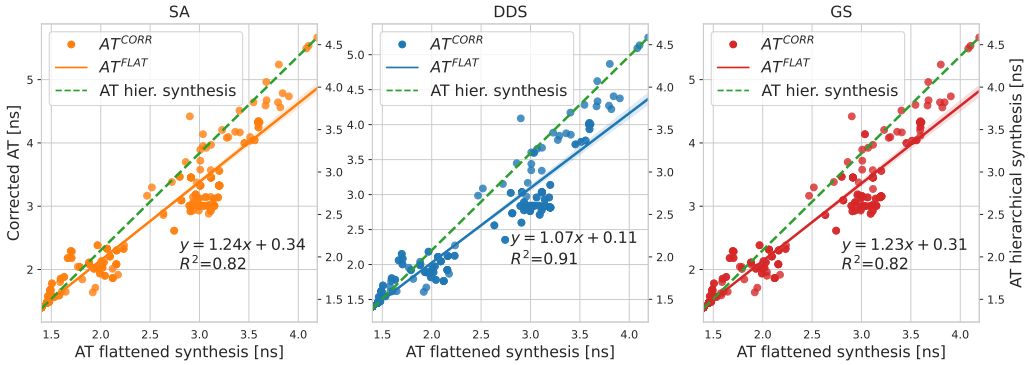
To evaluate the performance of the minimization task, the average values reported in Tables 9.10 and 9.11 are analyzed. Despite a slight decline in the averaged R^2 score, there is a noteworthy 0.07% reduction in the MAPE between $\mathbf{AT}^{\mathrm{CORR}}$ and $\mathbf{AT}^{\mathrm{FLAT}}$ before and after the minimization, indicating an overall decrease in the error. This reduction is statistically significant, given the many circuits (83) and primary outputs (574) evaluated. It is essential to note that error metrics are measured solely at the design endpoints, but realistic $\mathbf{AT}^{\mathrm{CORR}}$ values are computed for all pins in the RTL IR. The complexity of the minimization process depends on the design size and the number of delays to be corrected. An increase in the number of search iterations based on design complexity has the potential to enhance the correction results, albeit possibly leading to longer runtimes.

9.3.4. Runtime Analysis

The first two steps of the Fake Timer flow offer a notable advantage in terms of speed compared to the conventional flow involving EDA tools. For instance, when applied to the c17 benchmark example from Chapter 8, the Fake Timer only requires 1.77× the runtime of the conventional



(a) Industrial Designs



(b) ISCAS'85 Designs

Figure 9.10.: Minimization of $\mathbf{AT}^{\mathrm{CORR}}$ w.r.t. $\mathbf{AT}^{\mathrm{FLAT}}$ using different algorithms: simulated annealing (SA), dynamically dimensioned search (DSS), and greedy search (GS).

flow when executed on the same Linux machine equipped with an Intel® Xeon® CPU E5-2690 v4 at 2.60 GHz. Moreover, when averaging the runtimes across BKAs of 4, 8, 16, 32, and 64-bit widths, the first two steps of the Fake Timer are approximately two times faster than running the EDA tools.

The runtime required for solving the minimization task depends on the number of delays to correct and the maximum number of search iterations. For example, in the case of the c17 benchmark, correcting 24 delays using GS with 100 iterations and rerunning the block-based STA flow takes 41 seconds, rendering it 1.74 times slower compared to the conventional flow. It is essential to emphasize that the third step of the Fake Timer is optional and offers to back-annotate each pin in an RTL IR with accurate timing metrics, resembling the timing behavior of the flattened netlist. This process requires running flattened synthesis once, obtaining the worst ATs at the design endpoints, and solving the minimization task. Even though, this steps is slower than getting the timing reports from a flattened synthesized netlist, there is currently no alternative method to back-annotate all pins of an RTL IR with accurate timing information after flattening, which is important to perform the right RTL level optimizations. Future work could speed up this step by parallelizing the timing-graph traversal and the search iterations.

Table 9.11.: Error metrics for *corrected* ATs compared to the ATs derived from flattened synthesis using different optimization algorithms.

Regression error metric	Industrial Designs			ISCAS'85 Design			Average
	SA [164]	DDS [162]	GS [176]	SA [164]	DDS [162]	GS [176]	Best algorithms
R^2	0.56	0.59	**0.70**	0.81	**0.91**	0.82	**0.81**
MAPE [%]	17.71	19.03	**15.42**	6.64	**4.95**	6.46	**10.19**
ME [ns]	11.71	7.65	**9.01**	1.51	**1.18**	1.52	**5.10**

9.3.5. Summary and Conclusions

Fake Timer is the novel engine proposed in this thesis. It estimates pin-to-pin delays and slews to compute block-based ATs of RTL designs. Experimental results demonstrate a high correlation between computed ATs and hierarchical and flattened synthesis results, with coefficients of determination R^2 of 0.95 and 0.85, respectively. Furthermore, Fake Timer has the unique ability to *fake* or correct the predicted delays to minimize discrepancies between the computed ATs and the realistic outcomes from flattened netlists. This correction reduces the MAPE between the computed ATs and the results after flattening. Using corrected delays and ATs, more accurate RATs and slacks are back-annotated to all pins of the RTL IR. Fake Timer provides valuable timing metrics per each pin of the RTL IR without running RTL generation, synthesis, or STA tools. It is a unique method to estimate RTL designs delays, slacks, and critical regions even if logic synthesis flattens the design.

Throughout this thesis, many approaches have been developed to estimate timing metrics using ML. The advent of ML in digital design and timing analysis has been particularly noteworthy, attributable primarily to open-source EDA tools and advanced ML techniques such as LSTMs and GNNs. Table 9.12 compares the different contributions of this thesis with relevant works from the literature, which are comprehensively reviewed in Chapter 4. This table compares the design stage from which input features and ground-truth labels are drawn, and it lists the reported R^2 scores. Works by Guo et al. [136] and Ye et al. [135] exhibit a superior quality of predictions, i.e., higher R^2 scores, as they utilize gate-level netlists as inputs. From RTL, Sengupta et al. also achieve a high R^2 when estimating the Pareto-front between TNS and power.

The methods proposed in this thesis are pioneering in their use of an RTL IR as input and their estimation of pin-to-pin delay and slews, maximum ATs, or in performing an ML-enabled block-based STA, which, among other metrics, computes Arrival Times (ATs). Using an RTL IR as input makes the methods generalizable to any design.

Table 9.12.: R^2 score reported in relevant related works and each chapter of this thesis.

	Predicted Metric					
	ATs [136]	**ATs [135]**	**TNS [20]**	**Pin delays & slews (Chapter 6)**	**Maximum ATs (Chapter 7)**	**ATs Fake Timer (Chapter 8)**
Input Stage	Placement	Physical synthesis	RTL	RTL	RTL	RTL
Prediction Stage	Routing	Physical synthesis	Placement	Logic synthesis	Logic synthesis	Logic synthesis
Best R^2	0.90	0.99	0.95	0.87	0.82	0.95[a] 0.85[b] 0.81[c]

[a] Computed **AT** compared to **AT** after hierarchical synthesis.
[b] Computed **AT** compared to $\mathbf{AT}^{\mathrm{FLAT}}$ after flattened synthesis.
[c] Corrected $\mathbf{AT}^{\mathrm{CORR}}$ compared to $\mathbf{AT}^{\mathrm{FLAT}}$ after flattened synthesis.

In conclusion, Fake Timer reaches state-of-the-art R^2 scores and performs the first two steps around two times faster than the conventional flow. Although the third step of the Fake Timer flow, which is the delay correction, has higher runtimes, it provides a novel approach to back-annotate timing from flattened netlists to RTL blocks and to identify critical regions. Future work includes adding parallelization to speed up runtimes, improving the delay correction by running other optimization algorithms, and considering more component types and wire delays.

10. A Microarchitecture Search Use Case at RTL

This chapter outlines how the timing estimations from Chapters 7 and 8 can automate the search for optimal microarchitectures at early design stages. As depicted in Figure 1.8, the timing estimations serve as a guide for design-space exploration. If the estimations reveal a violation or sub-optimal results, the design is iteratively modified, and new estimations are obtained until an improved design is found. This process is particularly well-suited for RTL generation frameworks, which support automated and generic transformation mechanisms. Thus, instead of manually modifying the design, automated transformations can act on the RTL IR and generate equivalent functional but optimized microarchitectures. The primary benefit of this approach is the reduction in runtime, as RTL generation, synthesis, and STA runs are unnecessary. Same important is the availability of realistic times at RTL.

Given that, this thesis employs MetaRTL as an RTL generation framework, this chapter introduces additional examples of microarchitecture transformations implemented at the RTL IR. Some of these transformations need timing estimations as inputs to guide the microarchitecture selection. This guidance, or *microarchitecture search*, can be automated using (1) timing estimations as cost functions to minimize and (2) transformations as mechanisms to generate new points in the design space. As an example, this chapter presents how random search is utilized to find better MUX tree structures. Furthermore, it discusses how future research could leverage RL to further optimize this search process.

10.1. Microarchitectural Transformations

Section 2.2.1.2 defines the concept of model transformation within the scope of MDE. Building upon this definition, MetaRTL also offers a set of transformations, as explained in Section 2.2.3.3. In essence, model-to-model transformations modify the design's RTL IR, preserving the same functionality while modifying microarchitectural details. This section introduces three examples of model-to-model transformations implemented in MetaRTL. While all these transformations are design-independent, i.e., they can be applied to any input RTL IR, they are categorized as either *timing-independent* or *timing-dependent*.

Timing-independent transformations bear a resemblance to the optimization passes utilized in logic synthesis tools. The benefits of performing such transformations earlier, i.e., at the RTL IR, are twofold. First, synthesis tools still follow a *garbage in, garbage out* principle. Thus, the better the input RTL, the more optimized will be the resulting gate-level netlist [20], [137]. Additionally, it empowers hardware engineers to explore the design microarchitecture and possible modifications before setting up and running any synthesis tool. Similarly, timing-dependent transformations bring the same benefits. However, they need timing estimations

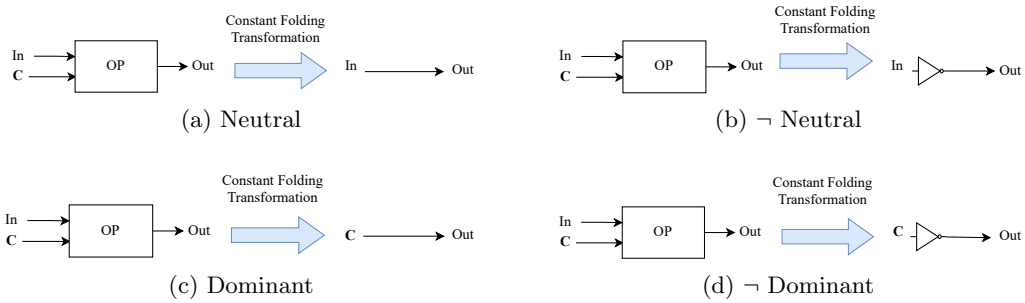


Figure 10.1.: Constant folding in logic operators (OP) with an input port (In) and a constant input (C).

as inputs to guide decisions about microarchitectural changes. In essence, model-to-model transformations at RTL IR emerge as a powerful tool for exploring different microarchitectures. The following describes the timing-independent transformations for constant folding on logic operators and MUXes. Lastly, the MUX tree decomposition is proposed as an example of timing-dependent transformations.

10.1.1. Constant Folding

Constant folding or propagation is a widely used optimization technique in which expressions or results are calculated beforehand to save execution time (in the software domain) or hardware components (in the hardware domain). Constant folding can be applied in both domains when there is a constant or literal input, such as a logic '0' or '1'. In MetaRTL, constant folding is applied at the RTL IR, targeting the logic operators and MUXes already introduced in Section 2.2.3.2. Notably, this transformation does not require timing estimations for guidance. Instead, it resembles the passes of logic synthesis tools performing Boolean optimizations.

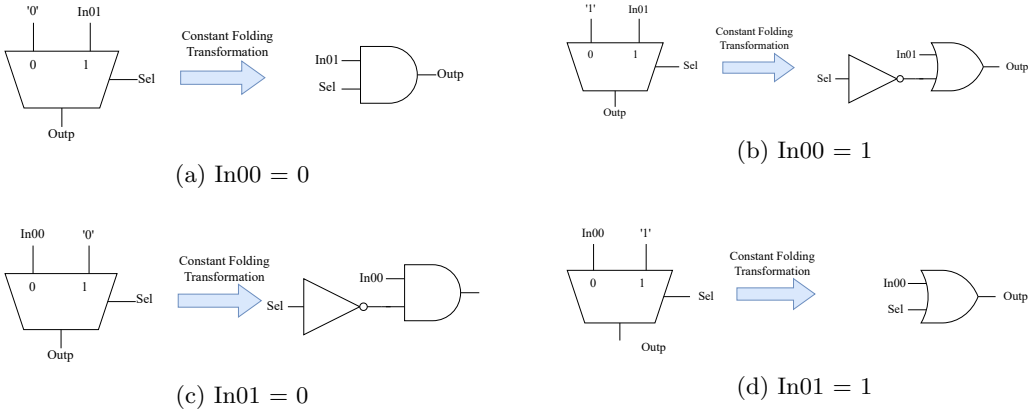
10.1.1.1. Logic Operators

In MetaRTL, logic operators encompass various types, including AND, OR, and XOR, in negated or bitwise, reduced, and logical forms. These logic operators can be transformed following four primary rules of constant folding listed in Figure 10.1. The dominant transformations imply that the constant value dominates the operation and impacts the result directly. Conversely, neutral transformations signify that the constant value is passive, with the final result being determined by the other input. The decision to apply the neutral or dominant transformation, or their negated forms, depends on the specific component type and whether the input constant value corresponds to a neutral or dominant value.

Table 10.1 shows the neutral and dominant values corresponding to each gate type. An automated transformation mechanism takes any RTL IR as input and visits every component. The constant folding transformation starts if a logic operator is found and one of its inputs is a constant. Subsequently, the constant input is compared with the respective neutral and

Table 10.1.: Dominant and neutral values per component type.

Operator Type	Neutral	Dominant	¬ Neutral	¬ Dominant
AND	1	0	✗	✗
OR	0	1	✗	✗
XOR	0	✗	1	✗
NAND	✗	✗	1	0
NOR	✗	✗	0	1
XNOR	1	✗	0	✗



(a) In00 = 0

(b) In00 = 1

(c) In01 = 0

(d) In01 = 1

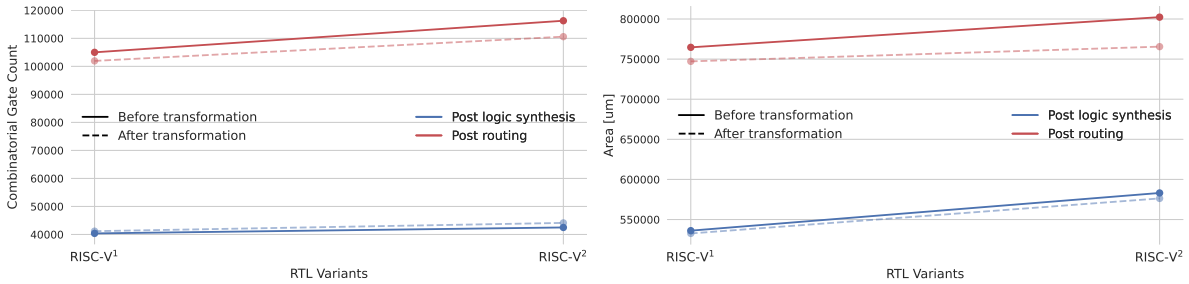
Figure 10.2.: Constant folding in 2-to-1 multiplexers.

dominant values based on the type of operator. In each scenario, a transformation is applied to replace the current logic component with a wire yielding only one of the inputs. Specific rules are adapted for each case of the logic operator to apply the transformations in a reduced, logical, or bitwise manner.

These transformations are applied across all logic operators in six slightly different RISC-V-based CPUs. To measure the benefit of the transformations, each Verilog-based RTL is generated for both the RTL IR before and after the application of the transformations. The average number of logic gates replaced per CPU is 30, and the worst AT gets reduced on average by 0.038%. As expected, the optimized RTL IRs yield slightly better metrics in terms of the number of combinatorial cells and ATs. The results are obtained using Yosys and OpenSTA under the same setup, operating conditions, and constraints. While the improvement in PPA is modest, considering the negligible cost of running this transformation, the impact remains noteworthy. Furthermore, the primary benefit of such transformations lies in their applicability to any input RTL IR. Additional benefits is a cleaner generated HDL code as parts without impact are removed from the design. Further, it improves the quality of timing estimation.

10.1.1.2. Multiplexers

A 2-to-1 MUX can be described by the Boolean Equation $Out = In00\neg Sel + In01Sel$. In the scenario where one of the inputs is a constant, specifically a logic '0' or '1', the 2-to-1 MUX



(a) Combinatorial gate count

(b) Area

Figure 10.3.: Results after applying constant folding in 2-to-1 MUXes inside two RISC-V-based CPU variants.

can be reduced to a maximum of two logic gates. There are four different possible cases of constant folding for 2-to-1 MUXes. Those are depicted in Figure 10.6. Further, a constant at the Selection pin makes the MUX disappear and be replaced by a wire to one input pin. The other input is not used. The connection driving this unused input can be simplified backwards further on. This transformation is applied to two different variants of an RISC-V core.

Figure 10.3 outlines the results per area and gate count after running logic synthesis with Yosys and routing with OpenROAD. The transformation is automatically applied to 80 and 143 MUXes in the RISC-V^1 and RISC-V^2 variants, respectively. The transformed RTL IRs exhibit less chip area than the golden designs. Notably, the positive impact of the transformation becomes even more significant after physical synthesis. Results after constant folding over MUXes and logic operators validate the assumption that logic synthesis tools are constrained by the *garbage in and garbage out* principle. Consequently, efforts to optimize the RTL IRs are not in vain; instead, they contribute to achieving a *right-at-first* RTL.

10.1.2. Multiplexer Tree Decomposition

Multiplexers can be found in many data paths of digital designs, varying in the number of input ports and bit widths. A MUX selects a data input given the selection input and forwards it to the output. An n-to-1 multiplexer has n data inputs, a selection input, and one output. The bit width of the selection input depends on the number of input pins $\lceil log_2(n) \rceil$. When described in HDL, MUXes are represented as *case* structures, where the inputs correspond to the possible values the output can take. At the RTL IR, MUXes are described as basic primitives with n inputs. During logic synthesis, MUXes are mapped to a set of logic gates or the MUXes available in the technology node. Generally, only 2-to-1 or 3-to-1 MUXes are defined. Thus, for MUXes in the RTL design with more than two or three inputs, a multiplexer tree is constructed. The approach for decomposing a larger MUX tree depends on the optimization passes of the synthesis tool. However, as evidenced by the constant folding transformations mentioned earlier, the representation of the MUX tree in the RTL design significantly influences the PPA metrics of the resulting gate-level netlist.

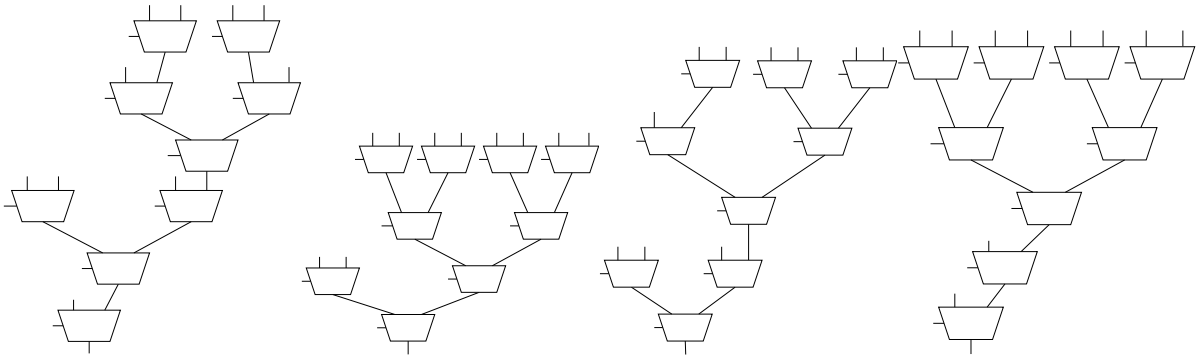


Figure 10.4.: Different decompositions of a 10-to-1 multiplexer tree.

Each n-to-1 MUX can be decomposed into an equivalent tree of 2-to-1 MUXes. The best tree structure, i.e., the best tree depth and order of inputs, highly depends on the ATs of the input and selection signals of the MUX. With a higher number of inputs for a MUX, the more options are available for restructuring the tree. The *balanced* tree and the *cascade* decompositions are the extreme cases for decomposing a tree multiplexer. The balanced tree decomposition resembles a fully balanced binary tree. A balanced tree provides the same path length from all data inputs to the output. However, in many cases, this decomposition is suboptimal, as the AT at the output is tied to the AT of the latest input. Additionally, in specific scenarios, extra 2-to-1 MUXes must be added to balance out the tree, increasing the area consumption. On the other hand, in the cascade tree, every 2-to-1 MUX takes a data input and the output of the previous MUX until the last MUX is connected to two data inputs. The resulting structure is a chain of depth $d = n-1$, where each node has only one child except for the last one. Regarding the ATs at the output, the cascade tree is more flexible than the balanced tree, as the input with the earliest or latest AT can be mapped to the first or last 2-to-1 MUX in the tree, respectively. In the cascade tree, $n-1$ selection signals are needed. Thus, an encoder is needed to map the original selection logic to the $n-1$ signals.

Figure 10.4 depicts other possible decompositions for a 10-to-1 MUX. In cases where the depth of the multiplexer tree is higher than the bit width of the selection input, an encoder is needed to preserve the multiplexing logic based on the original and new selection bits. For an n-to-1 multiplexer, the number of possible decompositions can be viewed as the number of structurally different binary trees with $n-1$ nodes. For a binary tree with n nodes, the number of different structures is given by the Catalan number [177] C_n in Equation 10.1. Consequently, the number of possible decompositions for an n-to-1 MUX tree is C_{n-1}. Moreover, each decomposition has a different impact on the AT at the output signal. Given this large number of possible decompositions, a number that rises exponentially with n, it is clear that choosing the optimal decomposition is a non-trivial task.

$$C_n = \frac{(2n)!}{(n+1)!n!} \tag{10.1}$$

The search for optimal MUX trees has motivated many works to optimize multiplexer trees for power [178], performance [179], and area [180]. In [178], the power consumption of a fully

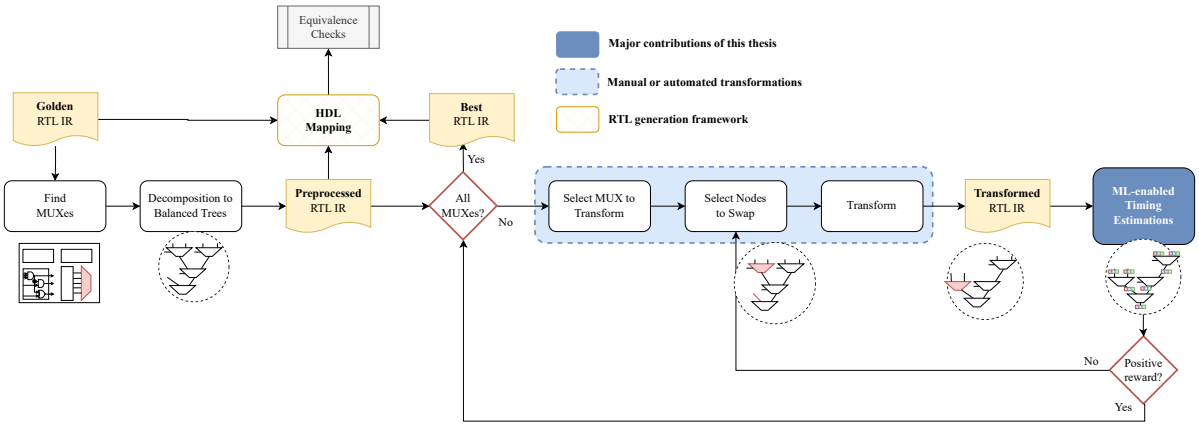


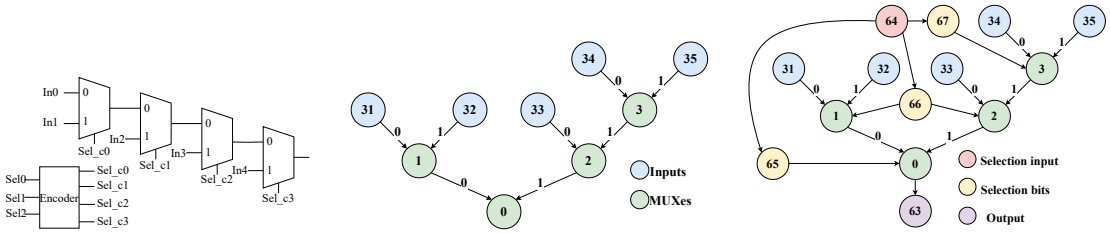
Figure 10.5.: Microarchitecture search flow at RTL IRs using transformations and the proposed ML-enabled timing estimations.

balanced MUX tree is minimized by using the probability of a certain data input being selected. In [179], MUX trees are decomposed based on the AT of the selection input. Finally, in [180], a GA is employed to optimize logic functions by using the minimum number possible of 2-to-1 MUXes.

10.2. Microarchitecture Search

The microarchitecture search involves two main components: Transformations to explore different RTL variants and timing information to guide the search. More powerful optimization can be achieved, especially using timing-dependent transformations. The necessary timing feedback can come directly from EDA tools, but this will require continuously running RTL generation, synthesis, and STA tools. This conventional flow is costly regarding runtime, as every RTL IR variant must be converted to its HDL description, synthesized, and analyzed. The proposed ML-enabled timing estimation engine takes RTL IRs as inputs and computes timing metrics per each component in the transformed RTL IR.

Figure 10.5 shows the basic workflow for a microarchitecture search for the multiplexer tree use case. First, a multiplexer component with n inputs is found in the *golden* RTL IR. This is decomposed as a balanced tree to use 2-to-1 multiplexer components available in the technology node. Afterward, a specific MUX is selected and its balanced tree can be transformed by modifying the order of the inputs or subtrees. The transformed RTL IR is the input to the ML-enable timing estimation engine, providing all relevant timing metrics per each pin in the tree. In particular, the ATs at inputs and outputs are decisive for the search. The former helps to find better tree structures, moving the latest input signals to the bottom of the tree. The latter guides the search. Trees with lower ATs or better slacks at the outputs are saved as best trees. These steps are repeated until the best tree is found for all MUXes in the golden RTL IR. As the original RTL IR is transformed multiple times in the flow, equivalence checks are performed to guarantee that the original and transformed designs have the same functionality.



(a) 5-to-1 MUX tree (b) Binary tree (c) Full graph

Figure 10.6.: Sample multiplexer tree and its graph representations.

The three steps encapsulated within the dashed blue box of Figure 10.5, involving the selection of a MUX for optimization, determining the modifications, and implementing the transformation in the RTL IR, constitute the pivotal stages of the microarchitecture search. These can be done manually per each RTL IR and each MUX. However, the process becomes inefficient due to the high number of designs, MUXes, and inputs per MUX. To address this challenge, optimization algorithms can be leveraged, varying in complexity from random search to more sophisticated approaches such as RL. Optimization algorithms are designed to identify the best solution to a problem that maximizes or minimizes an objective function. In the context of the multiplexer tree problem, the objective function can be formulated in terms of the computed ATs or slacks at the output of the tree before and after the transformations. These optimization algorithms also require the automatic exploration of different solutions in the search space, i.e., the automated selection of the next transformation to apply. The subsequent sections elaborate on the automation of the MUX tree transformation and the utilization of random search to automate the flow.

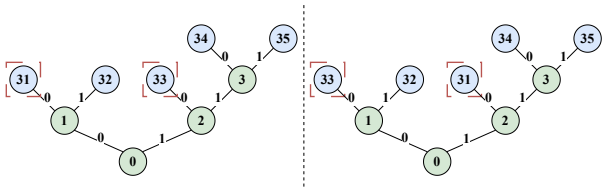
10.2.1. Automation of Transformations

As shown in Figure 10.5, the golden RTL IR is first preprocessed, finding and decomposing bigger MUXes on the design into an adequately balanced tree. To automate and generalize the transformation process for any MUX, the balanced tree is converted into a binary tree, providing the convenience of all MUXes being 2-to-1. In this representation, the MUXes and input ports serve as the nodes of the tree, each having a unique identifier number (ID). The connections between the different ports and the MUXes are represented using edges. Additional nodes are needed to describe the MUX tree as a graph structure, encompassing a node for the output port, a node for the selection port, and a node for each selection bit. Figure 10.6 illustrates a 5-to-1 MUX with its binary tree and complete graph representations.

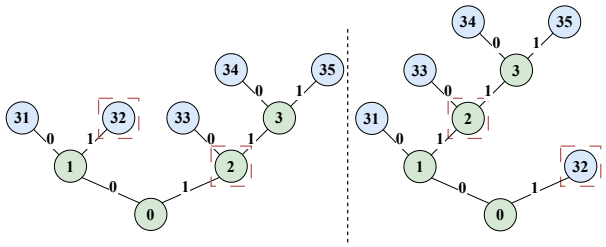
To ensure the MUX tree decomposition transformation can be universally applied to any MUX, regardless of the number of inputs, each type of node has a fixed ID range. With support for a maximum of 32 inputs, the ID ranges for the nodes representing MUXes span from [0, 30], being node 0, the tree's root, and with its output corresponding to the MUX output. The input nodes vary from [31, 62]. Additionally, the selection port is always identified with the ID 64 and is connected to all the selection bit nodes. Each selection bit controls a set of MUXes on the same tree level, with each selection bit possessing an ID from 65 plus the tree level.

Furthermore, Figure 10.6 also displays every node with its corresponding ID.

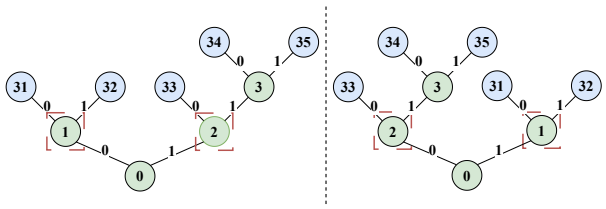
The transformation of the MUX tree translates into actions being applied to the binary tree. In order to enable an automated timing-dependent transformation flow, these actions shall be precisely defined, unique, and deterministic. Accordingly, this thesis implements the *swapping node* transformations, which involve moving a pair of nodes along with their sub-trees. In this process, the parent of the first node becomes the parent of the second node, and vice versa. After the nodes are swapped, the selection logic should be recalculated and fixed to preserve the functionality of the golden design. Given that there are three main types of nodes in the binary trees, three possible actions are identified: input-to-input, MUX-to-MUX, and MUX-to-input swapping. Figure 10.7 exemplifies the three types of actions for certain nodes within a 5-to-1 MUX.



(a) Swap(31, 33)



(b) Swap(2, 32)



(c) Swap(1, 2)

Figure 10.7.: Types of actions: (a) Input-to-input, (b) MUX-to-input, and (c) MUX-to-MUX

Nevertheless, not all actions swapping nodes are valid for all MUXes. For instance, swapping the duplicate node IDs will not change the tree's structure regardless of the node type. Moreover, illegal swaps target nodes that do not exist on the tree. Only if the MUX has the maximum number of inputs (e.g., 32), all node IDs will be available and possible to swap. Therefore, depending on the number of inputs of a MUX, some actions are excluded during runtime. The total number of possible swaps considering the maximum number of nodes in a tree with 64 inputs is $63 \times 63 = 3969$. As swapping actions are permutable, i.e., swapping nodes A and B is equivalent to swapping B and A, the total number of distinct actions be-

comes $\binom{63}{2} = 2016$. The specific number of valid transformations depends on each MUX and its number of inputs. As suggested by the Catalan number in Equation 10.1, the higher the number of inputs, the higher the number of possible tree structures. Moreover, every solution is affected by the different ATs at the inputs of the tree. Thus, automated search algorithms are needed to transform the MUX tree and retrieve the timing feedback.

10.2.2. Using Random Search

The easiest way to automate a design space exploration is to replace the manual application of transformations with a random search mechanism. Applied to the MUX decomposition task in Figure 10.5, the random search algorithm traverses all MUXes in the golden RTL IR, and per each MUX, it *randomly* selects a pair of nodes to be swapped in each iteration. This selection can be invalid if the pairs of nodes do not exist on the current MUX. Thus, a validation step is required every time after the node selection to avoid invalid transformations to a MUX. After a valid transformation is applied, timing feedback of the transformed RTL IR is required to quantify the benefit of the transformation. The current transformation is compared with the previous best tree in each iteration. If the timing benefit after the transformation increases, the current tree decomposition is saved as the best tree. This flow is repeated a maximum number of iterations. As a result, the best tree structure is employed in the final RTL IR.

The required timing feedback can be obtained from EDA tools. Thus, the transformed RTL IR is leveraged to generate its RTL, the corresponding gate-level netlist, and STA reports. From the latter, meaningful timing metrics are obtained, and a *reward* function is computed. As an alternative, ML-enabled timing estimations proposed in this thesis can be directly gathered from the RTL IR. The definition of the reward function shall be based on the hardware engineer's PPA optimization targets. The reward function in Equation 10.2 is employed for a timing-driven optimization. This considers only the worst-case ATs at the primary outputs of the design for both timing edges. Worst-case ATs can be easily found in STA reports. In this case, the transformations are evaluated in terms of their capabilities to optimize MUXes on the critical path of the design. When using timing estimations, the ATs at the output of the MUXes can be used instead. By doing so, the optimal tree structure is found for each MUX independently, whether this is on the critical path or not.

$$\text{Reward(action)} = \sum_{i \in \{\text{rise,fall}\}} 1 - \frac{AT\_\{i\}\_after\_action}{AT\_\{i\}\_golden} \tag{10.2}$$

This random search was run over four IP designs with different MUXes. When running 50 iterations per MUX, results show that random search is well-suited for small MUXes with up to four inputs. In this scenario, the possible number of transformations is reduced, and all possible combinations can be exhaustively explored within the 50 search iterations. As a result, the ATs of the critical paths of the whole IPs are improved. However, for larger MUXes with more than eight inputs, the random search struggled to find optimal structures that improve the critical path. With an increased number of inputs, the likelihood of the random search making progress through 50 iterations of valid actions diminishes. These results confirm the challenging nature of the problem and motivate further research in this domain.

10.2.3. Formulation as a Reinforcement Learning Task

More powerful search algorithms, such as RL, can be employed to optimize the MUX tree microarchitecture search. RL [181] is a subset of ML algorithms used to create agents capable of sequentially making decisions for solving complex problems. A typical RL task has two main entities: the agent and the environment. At each step of the algorithm, the agent fetches the state of the environment, denoted as s, and chooses an action a to be applied to the environment. The action taken by the agent modifies the state of the environment. This change is evaluated by the agent using a reward signal r. The agent aims to find a $\pi(s)$ policy, which depending on the current state, returns the action which maximizes the cumulative reward over the time. In this way, the agent learns optimal actions through trial and error. This makes RL particularly suitable for solving EDA problems, as it mimics the same sequential decision process usually done by hardware engineers. In recent years, RL has been applied to the EDA field at different stages of the digital design flow, such as macro placement [97], high-level synthesis [21], and microarchitecture optimization [182]. Inspired by those works, this thesis outlines the MUX tree decomposition as a task that an RL agent can solve.

The envisioned flow, using an RL agent, replaces the blue dashed box with an RL agent applying actions to the RTL IR and using the ML-enabled timing estimations as a reward. In each iteration, the RL agent chooses a MUX to optimize and samples an action to apply to it, i.e., a pair of nodes to be swapped. The action is passed alongside the RTL IR to MetaRTL to perform the transformation. The transformed RTL IR is then given as input to the ML-enabled timing estimation engine of Chapter 8 to get timing metrics. These are used to compute the reward function, similar to the random search. The reward function informs the RL agent about the effect of the applied action on the MUX's performance. Actions are applied to the MUX until a better tree structure is found. This cycle is repeated until all the transformed MUXes have been optimized. While the RL flow is not implemented in the scope of this thesis, it is a potential candidate for future research in the field.

11. Summary of Contributions

This thesis introduces well-defined methods for estimating timing metrics at the early stages of the ASIC design flow, using ML techniques as an alternative to traditional EDA tools. The methods in this thesis have been developed and evaluated using an RTL generation framework provided by Infineon Technologies AG. Moreover, the proposed methodologies and the key findings have been published in several scientific conferences [183]–[190], two journals [191], [192], and a book chapter [193].

The first step in this thesis is a review of the latest applications of ML in addressing EDA tasks. The advent of GNNs has been particularly significant, enabling a new realm of EDA applications where circuits, netlists, and layouts can be intuitively represented as graphs [184], [192]. This exhaustive review unveils two main aspects. First, there is a gap in the application of ML models during the early stages of the design. Second, primarily non-ML-based methods are applied to estimate the design metrics of an RTL. As such, this thesis is a trailblazer, introducing the use of ML techniques for timing estimation at RTL, being more robust and faster than traditional estimation methods. This thesis does not seek to replace conventional EDA tools with ML models. On the contrary, it aspires to incorporate ML into the prevailing flow and exploit the strengths of both methodologies.

The subsequent step in this thesis is establishing an automated data collection flow using an in-house RTL generation framework called MetaRTL. This framework generates thousands of random training designs covering different, valid, and specific configurations. It generates RTL IRs and Verilog-based RTL for each specification entry. Using this flow, training circuits and more complex evaluation circuits are generated. MetaRTL offers multiple advantages for an ML-based flow. A highlight feature is the intermediate representation (IR) of a design, which can be used to extract design features or to be optimized using *model-to-model* transformations [45].

EDA tools are incorporated into the proposed data generation to collect timing information and ground-truth labels for the designs. Significantly, this thesis is the pioneer involving open-source synthesis tools within an industrial development flow, analyzing the challenges and possible use cases [186]. The RTL generation framework is connected to OpenROAD, providing an automated specification-to-layout flow. This thesis analyses the use of OpenROAD against proprietary tools. Qualitative and quantitative comparisons are performed by analyzing the PPA performance obtained from both tools under similar designs and conditions [185]. The efficacy of the flow in this thesis is validated as a similar process emerges concurrently. In particular, the Robust Design Flow (RDF-2021) [142] incorporates Chisel, another RTL generation framework, and a data metrics collection system into OpenROAD to enable ML applications in the digital design flow. In contrast to that work, this thesis not only uses

the generated HDL for generating different IPs or RTL variants but also goes deeper into the exploration of the IR of the RTL generation framework for feature generation and potential optimizations.

Using the proposed data generation flow, the three significant contributions of this thesis are developed: the estimation of delays and slews of input-to-output connections inside components of an RTL IR [183], [187], [191], the estimation of the maximum AT among all its primary outputs [189], and an ML-enabled block-based STA engine called Fake Timer [190], [193].

This thesis formulates the pin-to-pin delay estimation as a regression task, solving it effectively with the use of ML models [183], [187], [191]. The implemented ML models enable the detection of critical pin connections of the RTL IR, even without generating the RTL. The presented methods interconnect RTL generation, open-source synthesis, and ML tools. This thesis carefully selects training circuits, features, and data structures to match the natural dynamics of the timing analysis flow.

- Training circuits are carefully selected to cover different configurations of the RTL IRs. Estimations do not target complete digital designs but the primitive blocks that constitute these designs. Thus, this thesis generates circuits using diverse primitive blocks with different configurations, such as logic operators and multiplexers [183] or an extended set adding comparators and arithmetic operators [191].

- Selected features describe pin-related information, such as bit widths and connections. Taking inspiration from the STA flow employing NLDMs, the slews or transition times are considered. Results revealed the slew to be essential to increase the prediction accuracy of the pin delays.

- Data structures are also carefully considered. Mapping RTL IRs to 1-D vectors results in tabular datasets, enabling the application of conventional ML techniques, such as shallow and deep regression models. This thesis recognizes that RTL IRs are intuitively graphs, and thus, modern ML methods such as GNNs can be explored [187]. Graph datasets are built, where each RTL IR is mapped to a featured graph and used as a training sample. Nodes and edges of the graphs are annotated with multidimensional features to exploit the capabilities of the GNNs even more.

Information about the delay through the pins of a component aids in identifying critical connections. However, global design metrics are indispensable for detecting critical regions or drawing informed conclusions about a design. These metrics include arrival times (ATs) or slacks. Hence, the second significant contribution of this thesis is the simultaneous estimation of pin delays, slews, and the worst-case ATs of an RTL IR [189]. GNN architectures are evaluated and employed to encode featured graphs and streamline the encoded embedding vectors to two regressors for edge and graph-level predictions. The complete pipeline is trained using a composed loss function. As a result, every pin of an RTL IR is annotated with delay and slew values, and the hardware designer gets an intuition of the worst-case arrival time at its primary outputs.

A design's worst-case AT helps compare RTL variants using different microarchitectures or HDL styles. However, a more robust analysis should be performed to pinpoint an RTL's critical components, regions, or paths. Thus, this thesis leverages the traditional block-based STA flow. Rather than approximating delays and slews from NLDMs in the technology, it utilizes ML predictions [193]. These are propagated through the timing graph representing an RTL IR. In this way, every block on the design is annotated with delays, slews, and maximum ATs, RATs, and slacks. The latter measures the block's criticality, which helps identify components that can be optimized further for area or timing. Moreover, nodes with the same worst slack compose the critical path.

Such a flow is invaluable for hardware designers, offering faster insights into an RTL IR without requiring RTL generation, synthesis, or STA. Even though results over different designs are promising, they correlate to the STA results when the designs are synthesized hierarchically. Hierarchical synthesis eases readability and debugging but limits the synthesis tool from performing cross-boundary optimizations. In other words, the estimations might not align with practical or realistic results. To address this, this thesis proposes the Fake Timer [190]. This consolidated STA estimation engine leverages pin-to-pin estimations to compute block-based timing metrics and aligns the computed metrics with results after flattened synthesis. Fake Timer is the main contribution of this thesis; it inputs an RTL IR and annotates each block with delays, slews, ATs, RATs, and slacks. It considers different timing arcs and resembles results after hierarchical and flattened synthesis. The identification of critical regions in the design is improved in this way.

Finally, this thesis analyses the microarchitecture search task as a potential use case where timing estimations of an RTL IR can offer valuable assistance. In this context, model-to-model transformations are implemented to facilitate the exploration of RTL microarchitectures. While the optimization of the RTL is typically done on the logic level, the implementation of constant folding for logic operators and multiplexers demonstrates that an optimized RTL input yields superior gate-level netlists. Moreover, the multiplexer tree restructuring is formulated as a timing-dependent transformation, where a set of actions or changes are applied to A multiplexer tree. The timing metrics predicted in this thesis are utilized as the reward function to optimize this tree.

In summary, the contributions of this thesis are:

- A comprehensive review of ML applications to the EDA flow, focusing on the RTL design stage and modern ML techniques, such as GNNs.
- An automatic flow for generating training and evaluation circuits employing an in-house RTL generation framework and open-source synthesis tools.
- Mapping the estimation of the delays of an input-to-output pin connection in a component of the RTL IR as a regression task to be solved using ML methods.
  - Training circuits target different configurations of primitive components, such as logic, arithmetic, multiplexing, and comparison operators.
  - Features match the nature of the task, especially considering input slews or transition times.

  - ML models such as shallow regressors, MLPs, and GNNs are employed to cope with tabular and graph-based datasets.

- Extension of the pin-to-pin delay and slew estimation to simultaneously predict the worst-case AT given an RTL IR. This global path-based timing metric allows the comparison among RTL variants with different microarchitectures or HDL coding styles.
- Consolidation of a timing estimation engine called Fake Timer. Fake Timer takes an RTL IR as input and annotates all its pins with timing metrics, such as delay, slews, ATs, RATs, and slacks. These highly correlate with STA results after hierarchical and flattened synthesis but without the execution of any EDA tool. Fake Timer pinpoints critical components in a design, which can be further optimized.
- Formulation of an RTL microarchitecture search task. Model transformations are leveraged as the steps of a design exploration mechanism, where the reward function to optimize is computed based on the timing estimated metrics proposed in this thesis.

Overall, this thesis has shown the potential of bringing current knowledge in ML into the hardware design realm. This thesis offers methods for timing estimation in RTL designs, with high accuracy and potential for integration into broader design flows. A distinguishing feature of the methodologies in this thesis is the use of RTL IRs as inputs and the analysis of their primitive components. These aspects enable the methods to generalize well to any RTL IR and to have a more exhaustive set of training and evaluation circuits compared to state-of-the-art approaches. Future work should extend the current flow to consider wire delays and extract ground-truth labels from post-placement or routing results. These labels would make the predictions more realistic, at the cost of higher runtimes during data collection. Moreover, the Fake Timer can be improved to increase the runtime efficiency for delay correction and the breadth of optimization algorithms and components it can manage.

Beyond the scope of this thesis, it is also possible to reflect on the impact of AI on the digital design flow. EDA tools have been developed to cover the design flow from various levels of abstraction, ranging from lumped-element models to system specifications. ML approaches offer additional layers of abstraction for EDA tasks. By learning from pattern distributions in massive training data sets, ML models reduce the complexity of the EDA tasks representing waveforms, circuits, and layouts in higher or lower-dimensionality vectors. However, unlike classical abstractions, it remains challenging to comprehend the nature of abstractions made by an ML model. Consequently, ML-based predictions cannot be fully explained or trusted (also regarding adversarial samples [194]). This lack of explainability or interpretability lowers the acceptance of ML-based solutions replacing classical methods in industrial flows.

According to Ren et al. [195], ML models still face challenges in solving complex tasks of the digital design flow unless the problem is confined to a smaller subset. This solution space is as limited as the collected training data are, as data can be biased or suffer from distribution shifts. Even with a proper data collection flow, as proposed in this thesis, allowing different RTL variants, and collecting the PPA results of each, no ML model can be trained to cover all ranges of features, all technology nodes, all target clock frequencies, or all possible design rules and constraints. Consequently, the ML-based timing estimations have demonstrated satisfactory accuracy when the problem is limited to a specific timing setup, component types,

and technology node and when considering the underlying physical models of the task, i.e., adding pin information, slew estimation, and block-based propagation.

The familiar aphorism *"All models are wrong, but some are useful"* from the statistician George E.P. Box [196] alludes to the limitations of abstraction models regarding the complexity of fundamental problems and how these models can still be helpful. In this sense, all ML models are wrong, too. While ML serves as an alternative for simplifying EDA tasks and achieving significantly good outcomes, there is still room for improvement in addressing complex EDA tasks [195]. Nevertheless, the aphorism from George Box emphasizes that despite their imperfections, some models can indeed be useful, particularly when managed adequately. In this context, this thesis and the proposed ML-based workflows excel in providing valuable early-timing feedback of RTL designs and helping find optimal microarchitectures.

Looking ahead, future research in ML for the digital design flow should emphasize the fusion of ML with classical optimization and heuristic algorithms in a coarse or fine-grain integration, as proposed by Ren et al. [195]. This integration of ML models and classical algorithms will enable the resolution of more complex tasks without constraining the solution space, ultimately leading to improved performance. It is also imperative that ML-based solutions are not confined to black-box implementations but instead leverage an understanding of the physical models underlying the task to select meaningful features and patterns from the training data. In the future, it is expected that reliable and interpretable ML models can solve core EDA tasks across various levels of abstraction.

Bibliography

[1] L. B. Mortensen, *Transforming your next IoT device with an ASIC*, en-GB, Nov. 2023. [Online]. Available: https://www.iotinsider.com/iot-insights/technical-insights/transforming-your-next-iot-device-with-an-asic/ (visited on 01/23/2024).

[2] *Mordor Intelligence provides market research*, en. [Online]. Available: https://www.mordorintelligence.com/market-analysis (visited on 01/23/2024).

[3] *Programmable Application Specific Integrated Circuit (ASIC) Market - Size, Share & Industry Analysis*, en. [Online]. Available: https://www.mordorintelligence.com/industry-reports/programmable-application-specific-integrated-circuit-asic-market (visited on 01/23/2024).

[4] G. E. Moore, "Cramming more components onto integrated circuits", *Electronics Magazin*, vol. 38, no. 8, 1965.

[5] C. A. Mack, "Fifty years of Moore's law", *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, no. 2, pp. 202–207, 2011. DOI: 10.1109/TSM.2010.2096437.

[6] M. M. Waldrop, "The chips are down for Moore's law", *Nature News*, vol. 530, no. 7589, p. 144, 2016. DOI: 10.1038/530144a. (visited on 01/23/2024).

[7] *16/12nm Technology - Taiwan Semiconductor Manufacturing Company Limited*, en. [Online]. Available: http://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_16_12nm (visited on 01/23/2024).

[8] D. Burg and J. H. Ausubel, "Moore's law revisited through Intel chip density", *PloS one*, vol. 16, no. 8, 2021. DOI: 10.1371/journal.pone.0256245.

[9] A. B. Kahng, "Invited: Reducing time and effort in IC implementation: A roadmap of challenges and solutions", in *Design Automation Conference (DAC)*, ACM/ESDA/IEEE, 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465871.

[10] I. T. R. for Semiconductors (ITRS). "The international technology roadmap for semiconductors 2001: Design". (2001), [Online]. Available: http://www.itrs2.net/itrs-reports.html.

[11] I. T. R. for Semiconductors (ITRS). "The international technology roadmap for semiconductors 2007: Design". (2007), [Online]. Available: http://www.itrs2.net/itrs-reports.html.

[12] I. T. R. for Semiconductors (ITRS). "The international technology roadmap for semiconductors 2013: System drivers abstract". (2013), [Online]. Available: http://www.itrs2.net/itrs-reports.html.

[13] D. MacMillen, R. Camposano, D. Hill, *et al.*, "An industrial view of electronic design automation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 19, no. 12, pp. 1428–1448, 2000. DOI: 10.1109/43.898825.

[14] *Wilson Research Group*. [Online]. Available: https://www.wilsonresearch.com/ (visited on 01/24/2024).

[15] H. D. Foster, "Trends in functional verification: A 2014 industry study", in *Design Automation Conference (DAC)*, ACM/EDAC/IEEE, 2015, pp. 1–6. DOI: 10.1145/2744769.2744921.

[16] H. D. Foster, "Trends in functional verification: A 2016 industry study", in *Design and Verification Conference and Exhibition Europe (DVCON)*, 2017, pp. 1–8.

[17] *The 2020 Wilson Research Group Functional Verification Study - Verification Horizons*, 2021. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2021/02/03/part-12-the-2020-wilson-research-group-functional-verification-study/ (visited on 01/24/2024).

[18] *The 2022 Wilson Research Group Functional Verification Study - Verification Horizons*, 2023. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2023/01/09/part-12-the-2020-wilson-research-group-functional-verification-study-2/ (visited on 01/24/2024).

[19] Synopsys, *Synopsys RTL Architect*. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/rtl-architect-ds.pdf (visited on 05/08/2023).

[20] P. Sengupta, A. Tyagi, Y. Chen, *et al.*, "How good is your Verilog RTL code? A quick answer from machine learning", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2022, pp. 1–9. DOI: 10.1145/3508352.3549375.

[21] N. Wu, Y. Xie, and C. Hao, "IronMan: GNN-assisted design space exploration in high-level synthesis via reinforcement learning", in *Great Lakes Symposium on VLSI (GLS-VLSI)*, ACM, 2021, pp. 39–44. DOI: 10.1145/3453688.3461495.

[22] R. L. Rudell, *Logic synthesis for VLSI design*. University of California, Berkeley, 1989.

[23] *Liberty user guides and reference manual suite*, 2013.

[24] *A guide to advanced process design kits*. [Online]. Available: https://semiengineering.com/a-guide-to-advanced-process-design-kits/ (visited on 07/01/2022).

[25] A. B. Kahng, J. Lienig, I. L. Markov, *et al.*, "Chip planning", in *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Dordrecht: Springer Netherlands, 2011, pp. 55–92. DOI: 10.1007/978-90-481-9591-6_3.

[26] A. B. Kahng, J. Lienig, I. L. Markov, *et al.*, "Global and detailed placement", in *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Dordrecht: Springer Netherlands, 2011, pp. 93–128. DOI: 10.1007/978-90-481-9591-6_4.

[27] A. Rajaram and D. Z. Pan, "Robust chip-level clock tree synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 30, no. 6, pp. 877–890, 2011. DOI: 10.1109/TCAD.2011.2106852.

[28] D. Müller, "Fast resource sharing in VLSI routing", Ph.D. dissertation, Universitäts-und Landesbibliothek Bonn, 2010.

[29] *Signoff-checks*. [Online]. Available: https://signoffsemiconductors.com/signoff-checks/ (visited on 07/01/2022).

[30] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: Constructing hardware in a Scala embedded language", in *Design Automation Conference (DAC)*, ACM, 2012, pp. 1216–1225. DOI: 10.1145/2228360.2228584.

[31] S. Takamaeda-Yamazaki, "PyVerilog: A Python-based hardware design processing toolkit for Verilog HDL", in *International Symposium Applied Reconfigurable Computing*, Springer, 2015, pp. 451–460. DOI: 10.1007/978-3-319-16214-0_42.

[32] J. Schreiner, R. Findenigy, and W. Ecker, "Design centric modeling of digital hardware", in *International High-Level Design Validation and Test Workshop (HLDVT)*, IEEE, 2016, pp. 46–52. DOI: 10.1109/HLDVT.2016.7748254.

[33] F. Truyen, "The fast guide to model driven architecture", *Cephas Consulting Corp*, 2006.

[34] S. Kent, "Model driven engineering", in *Integrated Formal Methods*, M. Butler, L. Petre, and K. Sere, Eds., Springer Berlin Heidelberg, 2002, pp. 286–298.

[35] W. Ecker and J. Schreiner, "Metamodeling and code generation", in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds. Dordrecht: Springer Netherlands, 2017, pp. 1–41. DOI: 10.1007/978-94-017-7358-4_32-1.

[36] J. Bézivin, "In search of a basic principle for model driven engineering", *Novatica/Upgrade*, vol. 5, Jan. 2004.

[37] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches", *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006. DOI: 10.1147/sj.453.0621.

[38] *Object Management Group (OMG)*. [Online]. Available: https://www.omg.org/ (visited on 01/03/2024).

[39] J.-M. Favre, "Foundations of model (driven) (reverse) engineering: Models", in *Language Engineering for Model-Driven Software Development*, J. Bezivin and R. Heckel, Eds., vol. 4101, Leibniz-Zentrum für Informatik, 2005, pp. 1–31. DOI: 10.4230/DagSemProc.04101.8.

[40] N. Koch, "Transformation techniques in the model-driven development process of UWE", in *International Conference on Web Engineering (ICWE)*, ACM, 2006, p. 3. DOI: 10.1145/1149993.1149997.

[41] H.-E. Eriksson, M. Penker, B. Lyons, *et al.*, *UML 2 toolkit*. John Wiley & Sons, 2003.

[42] E. R. Harold and W. S. Means, *XML in a nutshell: A desktop quick reference.* " O'Reilly Media, Inc.", 2004.

[43] N. Gerlin, E. Kaja, M. Bora, *et al.*, "Design of a tightly-coupled RISC-V physical memory protection unit for online error detection", in *International Conference on Very Large Scale Integration (VLSI-SoC)*, IFIP/IEEE, 2022, pp. 1–6. DOI: 10.1109/VLSI-SoC54400.2022.9939622.

[44] S. Prebeck, S. Ashok, M. Vaddeboina, *et al.*, "A scalable, configurable and programmable vector dot-product unit for edge AI", in *Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV)*, VDE, 2022, pp. 1–9.

[45] Z. Han, D. Wang, G. Rutsch, *et al.*, "Aspect-oriented design automation with model transformation", in *International Conference on Very Large Scale Integration (VLSI-SoC)*, IFIP/IEEE, 2021, pp. 1–6. DOI: 10.1109/VLSI-SoC53125.2021.9606984.

[46] V. B. Bavache, Z. Han, H. Hartlieb, *et al.*, "Automated SoC hardening with model transformation", in *Baltic Electronics Conference (BEC)*, IEEE, 2020, pp. 1–6. DOI: 10.1109/BEC49624.2020.9276994.

[47] D. Blaauw, V. Zolotov, and S. Sundareswaran, "Slope propagation in static timing analysis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 21, no. 10, pp. 1180–1195, 2002. DOI: 10.1109/TCAD.2002.802274.

[48] R. Hitchcock, G. Smith, and D. Cheng, "Timing analysis of computer hardware", 1, vol. 26, 1982, pp. 100–105. DOI: 10.1147/rd.261.0100.

[49] W. D. Cottrell, "Simplified program evaluation and review technique (PERT)", *Journal of Construction Engineering and Management*, vol. 125, no. 1, pp. 16–22, 1999. DOI: 10.1061/(ASCE)0733-9364(1999)125:1(16).

[50] *Synopsys timing constraints and optimization user guide*, version D-2010.03, 2010.

[51] "IEEE standard for integrated circuit (IC) delay and power calculation system", *IEEE Std 1481-1999*, 2000. DOI: 10.1109/IEEESTD.2000.7395044.

[52] TIOBE - The software quality company. "TIOBE Index for August 2024 - Python is chasing Java's TIOBE index records". (2024), [Online]. Available: https://www.tiobe.com/tiobe-index/ (visited on 07/13/2024).

[53] K. Devarajegowda, W. Ecker, and W. Kunz, "How to keep 4-eyes principle in a design and property generation flow", in *Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV)*, 2019, pp. 1–6.

[54] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from data*. AMLBook New York, 2012, vol. 4.

[55] A. L. Samuel, "Some studies in machine learning using the game of checkers", *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[56] J. Vamathevan, D. Clark, P. Czodrowski, *et al.*, "Applications of machine learning in drug discovery and development", *Nature Reviews Drug Discovery*, vol. 18, no. 6, pp. 463–477, 2019. DOI: 10.1038/s41573-019-0024-5.

[57] Y. Cao, S. Li, Y. Liu, *et al.*, "A comprehensive survey of AI-generated content (AIGC): A history of generative AI from GAN to ChatGPT", *arXiv preprint arXiv:2303.04226*, 2023.

[58] A. Qayyum, M. Usama, J. Qadir, *et al.*, "Securing connected and autonomous vehicles: Challenges posed by adversarial machine learning and the way forward", *Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 998–1026, 2020. DOI: 10.1109/COMST.2020.2975048.

[59] W. Wang and K. Siau, "Artificial intelligence, machine learning, automation, robotics, future of work and future of humanity: A review and research agenda", *Journal of*

Database Management (JDM), vol. 30, no. 1, pp. 61–79, 2019. DOI: 10.4018/JDM.2019010104.

[60] J. Hu and A. B. Kahng, "Invited paper: The inevitability of AI infusion into design closure and signoff", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2023, pp. 1–7. DOI: 10.1109/ICCAD57390.2023.10323684.

[61] J. H. Zar, "Spearman rank correlation", *Encyclopedia of Biostatistics*, vol. 7, 2005.

[62] A. Kraskov, H. Stögbauer, and P. Grassberger, "Estimating mutual information", *Physical Review E, Statistical, Nonlinear, and Soft Matter Physics*, vol. 69, no. 6, p. 66 138, 2004. DOI: 10.1103/PhysRevE.69.066138.

[63] E. Fix and J. L. Hodges, "Discriminatory analysis. Nonparametric discrimination: Consistency properties", *International Statistical Review*, vol. 57, no. 3, pp. 238–247, 1989. DOI: 10.2307/1403797.

[64] L. S. Shapley, *A value for N-person games*. Santa Monica: RAND Corporation, 1952. DOI: 10.7249/P0295.

[65] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions", in *International Conference on Neural Information Processing Systems (NIPS)*, Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 4768–4777. DOI: 10.5555/3295222.3295230.

[66] G. Dong and H. Liu, *Feature engineering for machine learning and data analytics*. CRC press, 2018.

[67] *Preprocessing data — scikit-learn 1.3.2 documentation*. [Online]. Available: https://scikit-learn.org/stable/modules/preprocessing.html (visited on 12/28/2023).

[68] X. Huang, A. Khetan, M. Cvitkovic, *et al.*, "TabTransformer: Tabular data modeling using contextual embeddings", *arXiv preprint arXiv:2012.06678*, 2020.

[69] X. Rong, "Word2vec parameter learning explained", *arXiv preprint arXiv:1411.2738*, 2014.

[70] J. Devlin, M. Chang, K. Lee, *et al.*, "BERT: Pre-training of deep bidirectional transformers for language understanding", in *North American Chapter of the Association for Computational Linguistics: Human Language Technologies NAACL-HLT*, vol. 1, Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/n19-1423.

[71] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains", in *International Joint Conference on Neural Networks (IJCNN)*, vol. 2, 2005, pp. 729–734. DOI: 10.1109/IJCNN.2005.1555942.

[72] G. Zhong, X. Ling, and L.-N. Wang, "From shallow feature learning to deep learning: Benefits from the width and depth of deep architectures", *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 1, e1255, 2019.

[73] S. K. Pal and S. Mitra, "Multilayer perceptron, fuzzy sets, classification", 5, vol. 3, IEEE Press, 1992, pp. 683–697. DOI: 10.1109/72.159058.

[74] M. H. Hassoun, *Fundamentals of artificial neural networks*. MIT Press, 1995.

[75] W. L. Hamilton, *Graph representation learning*. San Rafael, USA: Morgan & Claypool Publishers, 2020, pp. 1–159.

[76] L. Lovász, "Random walks on graphs: A survey", *Bolyai Society Mathematical Studies. Combinatorics, Paul Erdõs is Eighty*, vol. 2, no. 1-46, p. 4, 1993.

[77] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations", in *SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, New York, USA: ACM, 2014, pp. 701–710. DOI: 10.1145/2623330.2623732.

[78] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks", in *SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, New York, USA: ACM, 2016, pp. 855–864. DOI: 10.1145/2939672.2939754.

[79] J. Zhou, G. Cui, S. Hu, *et al.*, "Graph neural networks: A review of methods and applications", *AI Open*, vol. 1, pp. 57–81, 2020. DOI: https://doi.org/10.1016/j.aiopen.2021.01.001.

[80] C. Zhang, D. Song, C. Huang, *et al.*, "Heterogeneous graph neural network", in *SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, New York, USA: ACM, 2019, pp. 793–803. DOI: 10.1145/3292500.3330961.

[81] Y. Ma, Z. Guo, Z. Ren, *et al.*, "Streaming graph neural networks", in *SIGIR Conference on Research and Development in Information Retrieval*. New York, USA: ACM, 2020, pp. 719–728. DOI: 10.1145/3397271.3401092.

[82] M. Kampffmeyer, Y. Chen, X. Liang, *et al.*, "Rethinking knowledge graph propagation for zero-shot learning", in *Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2019, pp. 11 479–11 488. DOI: 10.1109/CVPR.2019.01175.

[83] R. Zhang, Y. Zou, and J. Ma, "Hyper-SAGNN: A self-attention based graph neural network for hypergraphs", in *International Conference on Learning Representations (ICLR)*, 2020, pp. 1–7.

[84] Z. Li, X. Shen, Y. Jiao, *et al.*, "Hierarchical bipartite graph neural networks: Towards large-scale E-commerce applications", in *International Conference on Data Engineering (ICDE)*, IEEE, 2020, pp. 1677–1688. DOI: 10.1109/ICDE48307.2020.00149.

[85] Z. Wu, S. Pan, F. Chen, *et al.*, "A Comprehensive survey on graph neural networks", *Transactions on Neural Networks and Learning Systems (TNNLS)*, vol. 32, no. 1, pp. 4–24, 2021. DOI: 10.1109/TNNLS.2020.2978386.

[86] L. Waikhom and R. Patgiri, "Graph neural networks: Methods, applications, and opportunities", *Computing Research Repository (CoRR)*, vol. abs/2108.10733, 2021.

[87] J. Bergstra, R. Bardenet, Y. Bengio, *et al.*, "Algorithms for hyper-parameter optimization", in *International Conference on Neural Information Processing Systems (NIPS)*, Curran Associates Inc., 2011, pp. 2546–2554. DOI: 10.5555/2986459.2986743.

[88] O. A. Montesinos López, A. Montesinos López, and J. Crossa, "Fundamentals of artificial neural networks and deep learning", in *Multivariate Statistical Machine Learning Methods for Genomic Prediction*. Springer International, 2022, pp. 379–425. DOI: 10.1007/978-3-030-89010-0_10.

[89] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. DOI: 10.5555/1953048.2078195.

[90] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in PyTorch", 2017.

[91] W. McKinney, "Data structures for statistical computing in Python", in *Python in Science Conference*, Austin, USA, vol. 445, 2010, pp. 51–56.

[92] M. Himsolt, "GML: A portable graph file format", Universitat Passau, Tech. Rep., 1997.

[93] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in *Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, USA, 2008, pp. 11–15.

[94] Y. Rong, T. Xu, J. Huang, *et al.*, "Deep graph learning: Foundations, advances and applications", in *SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, New York, USA: ACM, 2020, pp. 3555–3556. DOI: 10.1145/3394486.3406474.

[95] B. Komer, J. Bergstra, and C. Eliasmith, "HyperOpt-Sklearn", *Automated Machine Learning: Methods, Systems, Challenges*, pp. 97–111, 2019.

[96] G. Huang, J. Hu, Y. He, *et al.*, "Machine learning for electronic design automation: A survey", *ACM Transactions Design Automation Electronic Systems*, vol. 26, no. 5, 2021. DOI: 10.1145/3451179.

[97] A. Mirhoseini, A. Goldie, M. Yazgan, *et al.*, "A graph placement methodology for fast chip design", *Nature*, vol. 594, no. 7862, pp. 207–212, 2021. DOI: 10.1038/s41586-021-03544-w.

[98] Z. Xie, R. Liang, X. Xu, *et al.*, "Net2: A graph attention network method customized for pre-placement net length estimation", in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, ACM, 2021, pp. 671–677. DOI: 10.1145/3394885.3431562.

[99] A. Agnesina, S. S. K. Pentapati, and S. K. Lim, "A general framework for VLSI tool parameter optimization with deep reinforcement learning", in *Neural Information Processing Systems (NeurIPS), Workshop on ML for Systems*, Computer Science, 2020, pp. 1–6.

[100] Y.-C. Lu, S. Pentapati, and S. K. Lim, "The law of attraction: Affinity-aware placement optimization using graph neural networks", in *Proceedings of the 2021 International Symposium on Physical Design*, 2021, pp. 7–14. DOI: 10.1145/3439706.3447045.

[101] R. Kirby, S. Godil, R. Roy, *et al.*, "CongestionNet: Routing congestion prediction using deep graph neural networks", in *International Conference on Very Large Scale Integration (VLSI-SoC)*, IFIP/IEEE, 2019, pp. 217–222. DOI: 10.1109/VLSI-SoC.2019.8920342.

[102] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge", in *Design Automation Conference (DAC)*, 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465913.

[103] W. L. Neto, M. Austin, S. Temple, *et al.*, "LSOracle: A logic synthesis framework driven by artificial intelligence: Invited paper", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2019, pp. 1–6. DOI: 10.1109/ICCAD45719.2019.8942145.

[104] A. Hosny, S. Hashemi, M. Shalan, *et al.*, "DRiLLS: Deep reinforcement learning for logic synthesis", in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2020, pp. 581–586. DOI: 10.1109/ASP-DAC47756.2020.9045559.

[105] Y. Zhou, H. Ren, Y. Zhang, *et al.*, "PRIMAL: Power inference using machine learning", in *Design Automation Conference (DAC)*, ACM, 2019. DOI: 10.1145/3316781.3317884.

[106] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph neural network inference for transferable power estimation", in *Design Automation Conference (DAC)*, ACM/IEEE, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218643.

[107] W. Fang, Y. Lu, S. Liu, *et al.*, "MasterRTL: A pre-synthesis PPA estimation framework for any RTL design", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2023, pp. 1–9. DOI: 10.1109/ICCAD57390.2023.10323951.

[108] E. Ustun, C. Deng, D. Pal, *et al.*, "Accurate operation delay prediction for FPGA HLS using graph neural networks", in *International Conference on Computer-Aided Design (ICCAD)*, ACM, 2020. DOI: 10.1145/3400302.3415657.

[109] H. Mohammadi Makrani, F. Farahmand, H. Sayadi, *et al.*, "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design", in *Field Programmable Logic and Applications (FPL)*, 2019, pp. 397–403. DOI: 10.1109/FPL.2019.00069.

[110] L. Servadei, "Cost estimation and optimization of hardware/software co-designs using machine learning", Ph.D. dissertation, 2020.

[111] E. Zennaro, L. Servadei, K. Devarajegowda, *et al.*, "A machine learning approach for area prediction of hardware designs from abstract specifications", *Euromicro Conference on Digital System Design (DSD)*, pp. 413–420, 2018. DOI: 10.1109/DSD.2018.00076.

[112] R. Cheng and J. Yan, "On joint learning for solving placement and routing in chip design", in *Conference on Neural Information Processing Systems (NeurIPS)*, 2021, pp. 1–12. DOI: 10.5555/3540261.3541523.

[113] C.-K. Cheng, A. B. Kahng, S. Kundu, *et al.*, "Assessment of reinforcement learning for macro placement", in *International Symposium on Physical Design (ISPD)*, ACM, 2023, pp. 158–166. DOI: 10.1145/3569052.3578926.

[114] I. L. Markov, "The false dawn: Reevaluating Google's reinforcement learning for chip macro placement", *arXiv preprint arXiv:2306.09633*, 2023.

[115] L. Alrahis, A. Sengupta, J. Knechtel, *et al.*, "GNN-RE: graph neural networks for reverse engineering of gate-level netlists", IEEE, 2021, pp. 1–14. DOI: 10.1109/TCAD.2021.3110807.

[116] Y. Ma, H. Ren, B. Khailany, *et al.*, "High performance graph convolutional networks with applications in testability analysis", in *Design Automation Conference (DAC)*, ACM, 2019, pp. 1–6. DOI: 10.1145/3316781.3317838.

[117] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", in *Computer-Aided Verification*, 2010, pp. 24–40. DOI: 10.1007/978-3-642-14295-6_5.

[118] I. Sutherland, B. Sproull, and D. Harris, *Logical effort: Designing fast CMOS circuits*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[119] D. E. Wallace and M. S. Chandrasekhar, "High-level delay estimation for technology-independent logic equations", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE, 1990. DOI: 10.1109/ICCAD.1990.129876.

[120] J. Qiu, S. Reda, and S. Hassoun, "Fast, accurate a priori routing delay estimation", in *International Workshop on System Level Interconnect Prediction (SLIP)*, ACM/IEEE, 2010, pp. 77–82. DOI: 10.1145/1811100.1811119.

[121] M. Xu and F. J. Kurdahi, "Area and timing estimation for lookup table based FPGAs", in *Proceedings ED&TC European Design and Test Conference*, IEEE, 1996, pp. 151–157. DOI: 10.1109/EDTC.1996.494141.

[122] A. Macii, E. Macii, G. Odasso, *et al.*, "Regression-based macromodeling for delay estimation of behavioral components", in *Great Lakes Symposium on VLSI (GLS-VLSI)*, IEEE, 1999, pp. 188–191. DOI: 10.1109/GLSV.1999.757407.

[123] T. Koyagi, M. Fukui, and R. Saleh, "Delay macromodeling and estimation for RTL", in *International Symposium on Circuits and Systems (ISCAS)*, 2008, pp. 2430–2433. DOI: 10.1109/ISCAS.2008.4541946.

[124] A. Srinivasan, G. D. Huber, and D. P. LaPotin, "Accurate Area and Delay Estimation from RTL Descriptions", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1998. DOI: 10.1109/92.661259.

[125] M. Nemani and F. N. Najm, "Delay estimation VLSI circuits from a high-level view", in *Design Automation Conference (DAC)*, 1998, pp. 591–594.

[126] A. Fayyazi, S. Shababi, P. Nuzzo, *et al.*, "Deep learning-based circuit recognition using sparse mapping and level-dependent decaying sum circuit representations", in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 638–641. DOI: 10.23919/DATE.2019.8715251.

[127] L. Servadei, E. Mosca, E. Zennaro, *et al.*, "Accurate cost estimation of memory systems utilizing machine learning and solutions from computer vision for design automation", *Transactions on Computers*, vol. 69, no. 6, pp. 856–867, 2020. DOI: 10.1109/TC.2020.2968888.

[128] S. Ramesh and R. Jayaparvathy, "Artificial neural network model for arrival time computation in gate level circuits", *Automatika: časopis za automatiku, mjerenje, elektroniku, računarstvo i komunikacije*, vol. 60, no. 3, pp. 360–367, 2019. DOI: 10.1080/00051144.2019.1631568.

[129] T. Martin, G. Grewal, and S. Areibi, "A machine learning approach to predict timing delays during FPGA placement", in *International Parallel and Distributed Processing Symposium Workshops*, IEEE, 2021, pp. 124–127. DOI: 10.1109/IPDPSW52791.2021.00026.

[130] E. C. Barboza, N. Shukla, Y. Chen, *et al.*, "Machine learning-based pre-routing timing prediction with reduced pessimism", in *Design Automation Conference (DAC)*, ACM/IEEE, 2019, pp. 1–6.

[131] P. Sengupta, A. Tyagi, Y. Chen, *et al.*, "Early identification of timing critical RTL components using ML based path delay prediction", in *Workshop on Machine Learning for CAD (MLCAD)*, ACM/IEEE, 2023, pp. 1–6. DOI: 10.1109/MLCAD58807.2023.10299879.

[132] H. Ren, S. Nath, Y. Zhang, *et al.*, "Why are graph neural networks effective for EDA problems?", in *International Conference On Computer-Aided Design (ICCAD)*, 2022, pp. 1–8.

[133] P. Shrestha, S. Phatharodom, and I. Savidis, "Graph representation learning for gate arrival time prediction", in *Workshop on Machine Learning for CAD (MLCAD)*, 2022, pp. 127–133. DOI: 10.1145/3551901.3556478.

[134] Y.-C. Lu, S. Nath, V. Khandelwal, *et al.*, "Doomed run prediction in physical design by exploiting sequential flow and graph learning", in *International Conference On Computer-Aided Design (ICCAD)*, IEEE/ACM, 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643435.

[135] Y. Ye, T. Chen, Y. Gao, *et al.*, "Graph-learning-driven path-based timing analysis results predictor from graph-based timing analysis", in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, ACM, 2023, pp. 547–552. DOI: 10.1145/3566097.3567904.

[136] Z. Guo, M. Liu, J. Gu, *et al.*, "A timing engine inspired graph neural network model for pre-routing slack prediction", in *Design Automation Conference (DAC)*, 2022, pp. 1207–1212. DOI: 10.1145/3489517.3530597.

[137] Synopsys, *Synopsys unveils RTL architect to accelerate design closure*, en. [Online]. Available: https://www.prnewswire.com/news-releases/synopsys-unveils-rtl-architect-to-accelerate-design-closure-301023175.html (visited on 05/08/2023).

[138] W. Snyder, "Verilator and Systemperl", in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[139] T. Ajayi, V. A. Chhabria, M. Fogaça, *et al.*, "Invited: Toward an open-source digital flow: First learnings from the OpenROAD project", in *Design Automation Conference (DAC)*, ACM/IEEE, 2019, pp. 1–4.

[140] S. Hashemi, C.-T. Ho, A. B. Kahng, *et al.*, "METRICS 2.0: A machine-learning based optimization system for IC design", in *Workshop on Open-Source EDA Technology*, Computer Science, 2018, pp. 1–4.

[141] C. Wolf, J. Glaser, and J. Kepler, "Yosys-A free Verilog synthesis suite", 2013.

[142] J. Chen, I. H.-R. Jiang, J. Jung, *et al.*, "DATC RDF-2021: Design flow and beyond ICCAD special session paper", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2021, pp. 1–6. DOI: 10.1109/ICCAD51958.2021.9643553.

[143] *SKY130 PDK*. [Online]. Available: https://skywater-pdk.readthedocs.io/en/main/ (visited on 05/22/2022).

[144] *OpenSTA*. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenSTA (visited on 06/30/2022).

[145] A. B. Kahng, "A mixed open-source and proprietary EDA commons for education and prototyping", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2022, pp. 1–6. DOI: 10.1145/3508352.3561378.

[146] *Commit activity*. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenROAD/graphs/commit-activity (visited on 01/10/2024).

[147] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system", in *SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, ACM, 2016, pp. 785–794. DOI: 10.1145/2939672.2939785.

[148] *Pytorch-Widedeep*, Aug. 2021. [Online]. Available: https://github.com/jrzaurin/pytorch-widedeep (visited on 08/19/2021).

[149] A. F. Agarap, "Deep learning using rectified linear units (ReLU)", *arXiv:1803.08375*, 2018.

[150] M. Welling and T. N. Kipf, "Semi-supervised classification with graph convolutional networks", in *International Conference on Learning Representations (ICLR)*, 2017.

[151] J. Gilmer, S. S. Schoenholz, P. F. Riley, *et al.*, "Neural Message Passing for Quantum Chemistry", in *International Conference on Machine Learning (PMLR)*, 2017, pp. 1263–1272. DOI: 10.5555/3305381.3305512.

[152] K. Cho, B. Van Merriënboer, D. Bahdanau, *et al.*, "On the properties of neural machine translation: Encoder-decoder approaches", *arXiv preprint arXiv:1409.1259*, 2014.

[153] *Awslabs/dgl-lifesci*, Jan. 2024. [Online]. Available: https://github.com/awslabs/dgl-lifesci (visited on 01/12/2024).

[154] K. Kamiński, J. Ludwiczak, M. Jasiński, *et al.*, "Rossmann-toolbox: A deep learning-based protocol for the prediction and design of cofactor specificity in Rossmann fold proteins", *Briefings in Bioinformatics*, vol. 23, no. 1, 2021. DOI: 10.1093/bib/bbab371.

[155] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need", *Advances in Neural Information Processing Systems (NIPS)*, vol. 30, 2017. DOI: 10.5555/3295222.3295349.

[156] P. Velickovic, G. Cucurull, A. Casanova, *et al.*, "Graph attention networks", *stat*, vol. 1050, no. 20, pp. 10–48 550, 2017.

[157] L. Roeder, *Netron*, Oct. 2023. [Online]. Available: https://github.com/lutzroeder/netron (visited on 10/13/2023).

[158] F. Brglez, "A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN", in *International Symposium on Circuits and Systems (ISCAS)*, 1985.

[159] B. Li, "Hierarchical statistical static timing analysis considering process variations", Ph.D. dissertation, Technische Universität München, 2010.

[160] S. S. Sapatnekar, "Efficient calculation of all-pairs input-to-output delays in synchronous sequential circuits", in *International Symposium on Circuits and Systems (ISCAS)*, IEEE, vol. 4, 1996, pp. 520–523. DOI: 10.1109/ISCAS.1996.542015.

[161] *Bound-constrained optimization*, en-US. [Online]. Available: https://neos-guide.org/guide/types/bound/ (visited on 05/09/2023).

[162] B. A. Tolson and C. A. Shoemaker, "Dynamically dimensioned search algorithm for computationally efficient watershed model calibration", *Water Resources Research*, vol. 43, no. 1, 2007.

[163] S. Forrest, "Genetic algorithms", *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 77–80, 1996. DOI: 10.1145/234313.234350.

[164] D. Bertsimas and J. Tsitsiklis, "Simulated annealing", *Statistical Science*, vol. 8, no. 1, pp. 10–15, 1993.

[165] R. Solgi, *Genetic algorithm*, May 2023. [Online]. Available: https://github.com/rmsolgi/geneticalgorithm (visited on 05/18/2023).

[166] S. N., *Heuristic global optimization*, May 2023. [Online]. Available: https://github.com/sharma-n/global_optimization (visited on 05/18/2023).

[167] A. Ghaderi and J. Frounchi, "Power and speed analysis of CMOS-based multipliers using VEDIC techniques", vol. 11, 2016, pp. 62–68.

[168] C. S. Rani and C. Ramesh, "Design and implementation of high performance parallel prefix adders", 2014, pp. 5899–5903.

[169] J. Sklansky, "Conditional-sum addition logic", *IRE Transactions on Electronic Computers*, no. 2, pp. 226–231, 1960. DOI: 10.1109/TEC.1960.5219822.

[170] Brent and Kung, "A regular layout for parallel adders", *Transactions on Computers*, vol. 100, no. 3, pp. 260–264, 1982. DOI: 10.1109/TC.1982.1675982.

[171] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations", *Transactions on Computers*, vol. 100, no. 8, pp. 786–793, 1973. DOI: 10.1109/TC.1973.5009159.

[172] M. Balcilar, P. Héroux, B. Gauzere, *et al.*, "Breaking the limits of message passing graph neural networks", in *International Conference on Machine Learning (PMLR)*, 2021, pp. 599–608. DOI: 10.1145/3447548.3467373.

[173] E. Kaja, N. Gerlin, M. Vaddeboina, *et al.*, "Towards fault simulation at mixed register-transfer/gate-level models", in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–6. DOI: 10.1109/DFT52944.2021.9568310.

[174] *Verific Design Automation*. [Online]. Available: https://www.verific.com/ (visited on 05/21/2023).

[175] M. Hansen, H. Yalcin, and J. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering", *IEEE Design and Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999. DOI: 10.1109/54.785838.

[176] C. Sammut, "Greedy search", in *Encyclopedia of Machine Learning and Data Mining*. Springer US, 2017, pp. 596–596. DOI: 10.1007/978-1-4899-7687-1_353.

[177] R. P. Stanley, *Catalan numbers*. Cambridge: Cambridge University Press, 2015.

[178] U. Narayanan, H. W. Leong, K.-s. Chung, *et al.*, "Low power multiplexer decomposition", in *International Symposium on Low Power Electronics and Design (ISLPED)*, ACM, 1997, pp. 269–274. DOI: 10.1145/263272.263350.

[179] S. Thakur, D. F. Wong, and S. Krishnamoorthy, "Delay minimal decomposition of multiplexers in technology mapping", in *Design Automation Conference (DAC)*, ACM, 1996, pp. 254–257. DOI: 10.1145/240518.240565.

[180] R. James, P. Mythili, C. Anilkumar, *et al.*, "A modified technique for the optimal design of combinational digital circuits with multiplexers using genetic algorithm", in *International Conference on Computational Intelligence and Information Technology (CIIT)*, Institution of Engineering and Technology, 2013, pp. 225–229. DOI: 10.1049/cp.2013.2595.

[181] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey", *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996. DOI: 10.1613/jair.301.

[182] R. Roy, J. Raiman, N. Kant, *et al.*, "PrefixRL: Optimization of parallel prefix circuits using deep reinforcement learning", in *Design Automation Conference (DAC)*, ACM/IEEE, 2021, pp. 853–858. DOI: 10.1109/DAC18074.2021.9586094.

[183] D. Sánchez Lopera, L. Servadei, V. P. Kasi, *et al.*, "RTL delay prediction using neural networks", in *Nordic Circuits and Systems Conference (NorCAS)*, IEEE, 2021, pp. 1–7. DOI: 10.1109/NorCAS53631.2021.9599868.

[184] D. Sánchez Lopera, L. Servadei, G. N. Kiprit, *et al.*, "A survey of graph neural networks for electronic design automation", in *Workshop on Machine Learning for CAD (MLCAD)*, ACM/IEEE, 2021, pp. 1–6. DOI: 10.1109/MLCAD52597.2021.9531070.

[185] D. Sánchez Lopera, P. Kashyap, N. Gerlin, *et al.*, "Using open-source EDA tools in an industrial design flow", in *Design and Verification Conference and Exhibition Europe (DVCON)*, 2022, pp. 1–8.

[186] C. Lueck, D. Sánchez Lopera, S. Wenzek, *et al.*, "Industrial experience with open-source EDA tools", in *Workshop on Machine Learning for CAD (MLCAD)*, ACM/IEEE, 2022, pp. 143–143. DOI: 10.1109/MLCAD55463.2022.9900097.

[187] D. Sánchez Lopera and W. Ecker, "Applying GNNs to timing estimation at RTL", in *International Conference on Computer-Aided Design (ICCAD)*, ACM/IEEE, 2022. DOI: 10.1145/3508352.3561095.

[188] E. S. Alcorta, A. Gerstlauer, C. Deng, *et al.*, "Special session: Machine learning for embedded system design", in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2023, pp. 28–37. DOI: 10.1145/3607888.3608962.

[189] D. Sánchez Lopera, I. Subedi, and W. Ecker, "Using graph neural networks for timing estimations of RTL intermediate representations", in *Workshop on Machine Learning for CAD (MLCAD)*, ACM/IEEE, 2023, pp. 1–6. DOI: 10.1109/MLCAD58807.2023.10299859.

[190] D. Sánchez Lopera, R. N. Kunzelmann, E. Kaja, *et al.*, "Fake Timer: An engine for accurate timing estimation in register transfer level designs", in *International Symposium on Quality Electronic Design (ISQED)*, IEEE, 2024, pp. 1–8. DOI: 10.1109/ISQED60706.2024.10528723.

[191] D. Sánchez Lopera, L. Servadei, S. Prebeck, *et al.*, "Early RTL delay prediction using neural networks", *Microprocessors and Microsystems (MICPRO)*, vol. 94, 2022. DOI: https://doi.org/10.1016/j.micpro.2022.104671.

[192] D. Sánchez Lopera, L. Servadei, G. N. Kiprit, *et al.*, "A comprehensive survey on electronic design automation and graph neural networks: Theory and applications", *Transactions Design Automation Electronic Systems (TODAES)*, vol. 28, no. 2, 2023. DOI: 10.1145/3543853.

[193] D. Sánchez Lopera and W. Ecker, *"AI-enabled electronic circuit and system design - AI-enabled static timing analysis at early stages of the digital design flow"*, (ISQED), Ed. Springer- Reproduced with permission from Springer editor, just accepted.

[194] K. Liu, J. J. Zhang, B. Tan, *et al.*, "Can we trust machine learning for electronic design automation?", in *International System-on-Chip Conference (SOCC)*, 2021, pp. 135–140. DOI: 10.1109/SOCC52499.2021.9739485.

[195] H. Ren, B. Khailany, M. Fojtik, *et al.*, "Machine learning and algorithms: Let us team up for EDA", *IEEE Design and Test*, vol. 40, no. 1, pp. 70–76, 2023. DOI: 10.1109/MDAT.2022.3143427.

[196] G. E. Box and N. R. Draper, *Empirical model-building and response surfaces.* John Wiley & Sons, 1987.

[197] A. A. Ghazy and M. Shalan, "OpenLANE: The Open-Source Digital ASIC Implementation Flow", 2020.

[198] *The OpenROAD Project - Talk at DARPA Microelectronics Access Design Automation Meeting (MADAM)*, en-US, Jun. 2022. [Online]. Available: https://vlsicad.ucsd.edu/NEWS22/MADAM-OpenROAD-Presentation-ACTUAL.pptx (visited on 08/04/2022).

[199] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment", *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[200] M. Koefferlein, *KLayout layout viewer and editor*, 2018. [Online]. Available: https://klayout.de/ (visited on 05/18/2022).

[201] A. B. Chowdhury, B. Tan, R. Karri, *et al.*, "OpenABC-D: A Large-Scale Dataset for Machine Learning Guided Integrated Circuit Synthesis", *arXiv preprint arXiv:2110.11292*, 2021.

[202] A. Ghose, V. Zhang, Y. Zhang, *et al.*, "Generalizable cross-graph embedding for GNN-based congestion prediction", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643446.

[203] C. Tan, C. Xie, A. Li, *et al.*, "AURORA: Automated refinement of coarse-grained reconfigurable accelerators", in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1388–1393. DOI: 10.23919/DATE51398.2021.9473955.

[204] A. B. Kahng, "Advancing placement", in *International Symposium on Physical Design (ISPD)*, ACM, 2021, pp. 15–22. DOI: 10.1145/3439706.3446884.

[205] *RISC-V chip designed with open source tools eeNews Europe*, en-US, May 2022. [Online]. Available: https://www.eenewseurope.com/en/risc-v-chip-designed-with-open-source-tools/ (visited on 05/20/2022).

[206] *Design Automation Technical Committee — IEEE Council on Electronic Design Automation*. [Online]. Available: https://ieee-ceda.org/technical-committee/datc (visited on 09/06/2022).

[207] J. Jung, A. B. Kahng, S. Kim, *et al.*, "METRICS 2.1 and flow tuning in the IEEE CEDA Robust Design Flow and OpenROAD ICCAD special session paper", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643541.

[208] A. Chowdhury, B. Tan, R. Carey, *et al.*, *Too big to fail? Active few-shot learning guided logic synthesis*, 2022. DOI: 10.48550/ARXIV.2204.02368.

[209] H. Esmaeilzadeh, S. Ghodrati, J. Gu, *et al.*, "VeriGOOD-ML: An open-source flow for automated ML hardware synthesis", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2021, pp. 1–7. DOI: 10.1109/ICCAD51958.2021.9643449.

[210] A. B. Kahng, "Looking into the mirror of open source: Invited paper", in *International Conference on Computer-Aided Design (ICCAD)*, IEEE/ACM, 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942131.

[211] A. B. Kahng and T. Spyrou, "The OpenROAD project: Unleashing hardware innovation", in *Proc. GOMAC*, 2021.

[212] A. Benso, S. Di Carlo, G. Di Natale, *et al.*, "A watchdog processor to detect data and control flow errors", in *9th IEEE On-Line Testing Symposium (IOLTS)*, IEEE, 2003, pp. 144–148. DOI: 10.1109/OLT.2003.1214381.

[213] *Publications*. [Online]. Available: https://theopenroadproject.org/publications/ (visited on 09/07/2022).

[214] H. Liew, D. Grubb, J. Wright, *et al.*, "Hammer: A modular and reusable physical design flow tool: Invited", in *Design Automation Conference (DAC)*, ACM/IEEE, 2022, pp. 1335–1338. DOI: 10.1145/3489517.3530672.

Appendices

A. Details of Design Specifications

Table A.1.: Configuration details used for training one model per component type in Chapter 8.

Comp. Category	Comp. Type	# Hidden Layers	# Input Features	# Trainable Parameters	Learning Rate α	Optimizer	Loss Function
Logic	BAND	4	14	80,253	0.002	Adam	MSE
	BNAND	4	14	38,851	0.007	Adam	MAE
	BOR	4	14	28,039	0.003	Adam	MAE
	BNOR	4	14	24,216	0.007	Adam	MSE
	BXOR	4	14	60,653	0.011	Adam	MSE
	RAND	2	14	40,015	0.0094	Adam	MAE
	RNAND	3	14	33,255	0.0042	RMSprop	MSLE
	RNOR	1	14	23,300	0.0165	RMSprop	MSLE
	ROR	4	14	68,928	0.022	Adam	MSE
	RXOR	2	14	22,292	0.0042	RMSprop	MSE
	LAND	4	14	56,476	0.0011	Adam	MAE
	LNAND	2	14	38,055	0.0125	Adam	MSE
	LOR	4	14	62,096	0.012	Adam	MAE
	LXOR	2	14	45,968	0.003	RMSprop	MSLE
	NOT	4	12	59,261	0.0015	RMSprop	MSE
Branch	EQ	4	13	60,567	0.0018	Adam	MSE
	NEQ	2	13	53,475	0.0078	Adam	MSE
	GTEQ	4	13	47,272	0.0087	Adam	MAE
	GT	2	13	30,219	0.0009	RMSprop	MSLE
	LT	2	13	38,207	0.009	Adam	MSE
	LTEQ	4	14	24,533	0.0026	RMSprop	MSLE
Multiplexing	MUX	4	14	25,788	0.0043	Adam	MAE
	DEMUX	4	14	61,822	0.013	Adam	MSE
Arithmetic	HWPLUS	4	14	80,253	0.0021	Adam	MSE
	HWMINUS	4	14	74,746	0.0054	Adam	MSE
	HWMUL	5	14	51,665	0.0076	RMSprop	MSE

Table A.2.: Restricted specifications based on configurations used on sample complex designs.

Comp. Category	Comp. Type	# Inputs	Inputs Bit width	Fan-out
Logic	BAND	[1, 6], 32	[1, 4], 8, 16, 24, 26, 31, 32	[0, 31]
	BNAND	2	1	1
	BOR	[1, 32]	1, 2, 32	[0, 10], [15, 19]
	BNOR	2	1	1, 2
	BXOR	2, 10, 12, 14, 16, 18, 20, 22, 24	1, 8, 32	[1, 5]
	LAND	[1, 8]	1, 8, 32	[0, 12], 16, 17, 20, 32
	LNAND	2	1	1
	LOR	[1, 9]	1, 8, 32	[0, 7]
	LXOR	2	1	1
	RAND	1	2, 32	1
	RNAND	1	2	1,2
	ROR	1	[2, 10], [15, 18], 31, 32, [41, 51], [55, 59], 63, 76	[1, 5], 7, 15
	RNOR	1	2	1
	RXOR	1	2, 4, 8, 9, 26, 31, 32	1, 27
	NOT	1	[1, 4], 8, 9, 16, 18, 32, 34, 64	[1, 4], 31, 32
Comparators	EQ	2	[1, 14], 16, 18, 32	[0, 8], 10, 14, [32, 34]
	NEQ	2	[1, 8], 32	1, 2
	GTEQ	2	[1, 5], [9, 12], [16, 18], 21, 31, 32	[1,3]
	GT	2	[3, 10], 12, 14, 16, 18, 20, 32	[1,3]
	LTEQ	2	2, 10, 14, 16, 17, 32	[1, 3]
	LT	2	[2, 5], [10, 12], 16, 18, 19, 30, 32	[1, 4]
Multiplexing	MUX	[1, 21], 23, 26, 31, 32, 38, 44, 63, 64, 128, 256	[1, 12], [14, 20], [30, 37], 41, 42, 49, 64, 65, 68, 72, 80, 88	[0, 8], 10, 12, 18, 19, 21, 22, 24, 26, 32, 38, 43, 62, 64
	DEMUX	1	[1, 3], 8, 16, 20, 32, 88	[0,3]

Continued on next page

Table A.2 – *Continued from previous page*

Comp. Category	**Comp. Type**	**# Inputs**	**Inputs Bit width**	**Fan-out**
Arithmetic	HWPLUS	2, 3, 8	$[1, 10], 12, [16, 22]$, $[30, 32]$, 34, 36, 64	[1, 4], 9
	HWMINUS	2	[1, 12], [16, 21], 31, 32	[1, 4] , 7
	HWMUL	2	2, 9, [16, 19], 24, 32	[1, 7]

B. Papers in the Scope of this Thesis

During the course of this thesis, open-source tools to gather training data were studied. These studies result in papers that are summarized in this appendix.

B.1. Industrial Experience with Open-Source EDA Tools

Commonly, the design flow of integrated circuits from initial specifications to fabrication employs commercial, proprietary EDA tools. While these tools deliver high-quality, production-ready results, they can be seen as expensive black boxes and thus, are not suited for research and academic purposes. Innovations on the field are mostly focused on optimizing the quality of the results of the designs by modifying core elements of the tool chain or using techniques of the ML domain. In both cases, researchers require many or long runs of EDA tools for comparing results or generating training data for ML models. Using proprietary, commercial tools in those cases may be either not affordable or not possible at all.

With OpenROAD[1] ("Foundations and Realization of Open Accessible Design") [139] and OpenLane[2] [197] mature open-source alternatives emerged in the past couple of years. The development is driven by a growing community that is improving and extending the tools daily [198]. In contrast to commercial tools, OpenROAD and OpenLane are transparent and allow inspection, modification and replacement of every tool aspect. They are also free and therefore are well suited for use cases such as ML data generation. Specifically, the fact that no licenses are needed neither for the tools nor for the default PDKs enables even fresh students and starters on the field to quickly deploy their ideas and create initial proof of concepts.

Therefore, we at Infineon are using OpenROAD and OpenLane for more experimental and innovative projects. Our vision is to build initial prototypes using free software, and then improve upon them by cross-checking and polishing with commercial tools before delivering them for production.

The first steps involved getting OpenLane installed and running in our company IT infrastructure. While their developers offer convenient build methods using Docker [199] containers, these cannot be used in Infineon's compute farm. This, and also the fact that most of the open-source tools are currently evolving quickly with little to no versioning, lead to the setup of an in-house continuous integration and continuous delivery system for nightly and weekly builds of the tools. Once the necessary tools were installed and running, effort was put into integrating Infineon's in-house technology data, since OpenROAD/OpenLane are per default making use of open-source PDKs such as the SKY130[3] [143] PDK from SkyWater and Google.

At Infineon, we envision two use cases for OpenROAD/OpenLane: physical synthesis hyper-parameter exploration (and tuning) and optimization of the complete flow starting from RTL.

[1] https://github.com/The-OpenROAD-Project
[2] https://github.com/The-OpenROAD-Project/OpenLane
[3] https://github.com/google/skywater-pdk

First, our goal is to use OpenROAD's AutoTuner in the path-finding phase to automatically and cost-effectively find optimal parameters for the flow and then build upon these results within a commercial tool for the later steps near the tapeout. Second, we want to include not only the synthesis flow inside the optimization loop of the AutoTuner, but also our in-house RTL generation framework (MetaRTL [32]). For instance, having RTL generators for a RISC-V CPU and also the results of simulated runtime benchmarks for each iteration, the AutoTuner should be able to change fundamental aspects (for example number of pipeline stages) of the RTL to reach certain power, performance, and area requirements when running the benchmark code on the CPU.

Overall, we see OpenROAD/OpenLane as a viable alternative to commercial tools, especially for research and academic use, where modifications to the tools are needed and where very long and otherwise costly tool runtimes are expected.

B.2. Using Open-Source EDA Tools in an Industrial Design Flow

B.2.1. Abstract

Commonly, semiconductor manufacturers employ commercial electronic design automation tools to get production-ready results. These tools are costly and closed source. Thus, they are not suited for academic and research purposes. Seminal works have presented open-source alternatives for stages of the design flow. However, executing standalone tools is time-consuming, and iterations of the design flow are costly. To overcome this, an open-source toolchain for digital layout is presented: OpenROAD. Even though OpenROAD has been described extensively in the literature, its performance w.r.t. commercial tools has not been evaluated. To address this, different implementations of RISC-V designs using an in-house hardware generation framework have been generated. Those designs are synthesized using OpenROAD and a commercial tool and mapped to a commercial and open-source technology library. Experiments compare optimization capabilities, runtimes, and the Quality of Results (QoRs) along the design flow. Commercial tools provide higher QoRs, lower runtimes, and better optimization capabilities. However, OpenROAD represents a good trade-off between performance, license costs, framework transparency, and support to the academic community. The comparison of OpenROAD and commercial tools is relevant to analyze the validity of machine learning models trained with OpenROAD outcomes, and to promote the use of open-source tools.

B.2.2. Introduction

Typically, the chip design flow from initial specification to Graphic Data System (GDS) format is conducted using commercial EDA tools to obtain high QoRs and production-ready layouts. However, commercial tool licenses are expensive. Thus, they are not appropriate for academic and research purposes, which need many and long tool runs. Even though commercial tool vendors provide support to customers, their tools are black boxes, and innovation in the design flow itself is constrained. To overcome these limitations, seminal papers have introduced EDA open-source tools, such as ABC [117] and KLayout [200].

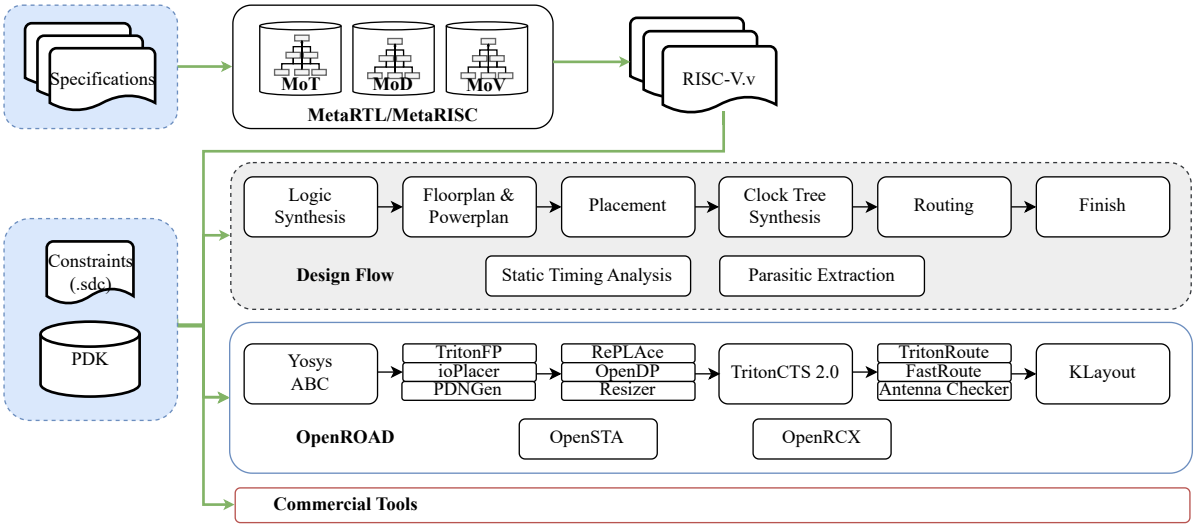


Figure B.1.: Industrial design flow from specifications to layout.

The digital design flow applies the "divide and conquer" strategy, going from specifications to layouts. Nevertheless, using standalone tools is time-consuming, and design flow iterations are costly. To overcome these issues, a DARPA IDEA program presented in 2018 the OpenROAD[4] project [139]: an open-source toolchain for digital layout aiming to run 24 hours with no human-in-the-loop. OpenROAD interconnects standalone open-source tools to build the design flow from RTL to GDS. Figure B.1 summarizes the stages of the design flow and their respective open-source tool alternatives.

OpenROAD popularity is growing and is being used inside flow controllers such as OpenROAD-flow-scripts[5], OpenLane[6] [197], and the Robust Design Flow (RDF)-2021[7] [142]. These flow controllers use OpenROAD as the main building block and connect it with other tools to add optimizations, improve design exploration or automate data collection. The OpenROAD flow controllers bring many benefits: adjustable source code, easy installation, and open exchange with the community. For research and academia, OpenROAD contribution is even higher. Innovation is made through OpenROAD code extension and modification, but also through ML models learning from OpenROAD-generated objects and reports [136], [201]–[203].

Even though the number of ML applications using OpenROAD outcomes continues to increase, to the best of our knowledge, this work is the first to compare the performance of OpenROAD w.r.t. the commercial tools. To solve this, this paper compares OpenROAD with a commercial alternative within the industrial design flow setup shown in Figure B.1. The main contributions of this paper are:

- Different RISC-V implementations are generated using an in-house RTL generator.

[4]https://github.com/The-OpenROAD-Project/OpenROAD
[5]https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts
[6]https://github.com/The-OpenROAD-Project/OpenLane
[7]https://github.com/ieee-ceda-datc/datc-rdf

- A proprietary and an open-source PDK are used.
- Experiments compare the QoRs along the design flow, runtimes, and optimization capabilities.

Results confirm that the QoRs of commercial tools is higher. However, OpenROAD represents a good trade-off between QoRs, costs, and framework transparency. OpenROAD is an enabler of research in EDA to foster new algorithms in the design flow and to collect ML training data with distributions not so far from the obtained with commercial tools.

This part of the appendix is organized as follows: Section B.2.3 surveys related works using OpenROAD. Section B.2.4 describes the industrial design flow used. The different implementations of the RISC-V are presented in Section B.2.4.4. The synthesis results using commercial and open-source tools are shown in Section B.2.5. Finally, Section B.2.8 concludes by summarizing the results and mentioning future directions.

B.2.3. Related Work

In the following, we mention state-of-the-art works already using OpenROAD.

As OpenROAD reduces design efforts and allows a better design-space exploration, the number of studies using this toolchain is increasing. For instance, in [204], the future research directions on placement are linked to the availability of open-source tools and PDKs, such as OpenROAD and the SkyWater Technology Foundry [143]. OpenROAD is also being used by the OpenLane flow and the RDF-2021. OpenLane is an open-source RTL-to-GDSII flow developed by Efabless. It adds custom tools for better design space exploration. Thus, it is a valuable tool for academic projects, such as developing a RISC-V 32-bit processor [205]. The RDF-2021 flow developed by the IEEE CEDA Design Automation Technical Committee (DATC) [206] is a framework for EDA-flow-related research. The latest RDF version includes Chisel/FIRRTL [30], a hardware generation tool built in Scala, and METRICS2.1 [207], a metric system to collect data and enable ML applications.

OpenROAD opens the door to ML applications optimizing tool configurations and learning from tool outcomes [103], [136], [201]–[203], [208], [209]. LSOracle [103] has been added as a plugin to Yosys to optimize logic mapping. In [201], a dataset called OpenABC-D is built to predict netlists QoRs for given tool parameters. OpenABC-D is leveraged by the Bulls-Eye [208] framework, which predicts the quality of synthesized netlist and includes simulated annealing to search the design space. Similarly, VeriGOOD-ML [209] generates Verilog code and validates it using OpenROAD outcomes. OpenROAD is used to build datasets for predicting design metrics as congestion [202], arrival times and slack [136], and PPA of Coarse-grained Reconfigurable Arrays (CGRAs) [203].

OpenROAD published papers describe the advances of the open-source flow and some attempts of benchmarking [207], [210], [211]. In [207], the AutoTuner framework finding optimal OpenROAD parameters is evaluated over three designs and two different SkyWater libraries.

Table B.1.: Brief comparison of used PDKs

PDK	License Type	PVT			# Lines Lib. File [$\times 10^3$]	# Standard Cells
130 nm	open-source	1.0 P	1.80 V	25°C	333.5	753
40 nm	proprietary	1.3 P	1.08 V	-40°C	14678.9	852

In [210], the number of commits and citations are proposed as metrics to overcome the challenge of open-source tools' progress quantification. A first comparison was conducted between a commercial and the OpenROAD placer. Similar to [210], [211] discusses design successes from OpenROAD and community engagement. They compare the capacitance and resistance values provided by OpenRCX and OpenSTA with results from commercial parasitic extraction and timing analysis tools. Even though OpenROAD has been described and used extensively in the literature, to the best of our knowledge, no studies compare its performance w.r.t. commercial tools.

B.2.4. Industrial Design Flow from Specifications to GDS

This section describes the industrial setup shown in Figure B.1.

B.2.4.1. Specification-to-RTL Flow

As recognized in [142], the flow controllers using OpenROAD highly benefit from hardware generation frameworks, which generate different designs and microarchitectures. In the RDF-2021, Chisel/glsfirrtl is incorporated. This paper employs an in-house generation framework called *MetaRTL* [32]. MetaRTL uses meta-modeling and applies the MDA principle to RTL generation [33]. In general, this framework provides three layers of a design: the initial specifications or Model of Things (MoT), the intermediate representation or Model of Design (MoD), and finally, the mapping to an HDL or Model of View (MoV). When integrated into OpenROAD, MetaRTL benefits are twofold: Python is the programming language used for hardware generation, OpenROAD flow controllers, and ML applications. Second, formal properties are generated, and it is possible to verify MetaRTL designs following a four-eyes principle [53]. The formal verification capabilities of our in-house framework motivate even further a complete flow from specification to GDS. However, formal verification is beyond the scope of this paper.

B.2.4.2. RTL-to-GDS Flow

This paper compares a commercial tool and the flow controller OpenROAD-flow-scripts. Figure B.1 shows the open-source tools within OpenROAD. Yosys **yosys** and ABC perform logic synthesis and are built standalone. Yosys is an open-source Verilog synthesis tool that provides

Table B.2.: Features of the generated and synthesized RISC-V designs.

Design	Extension units	No. Lines of code	No. Components	No. Input bits	No. Output bits
RISC-V[1]	CRC, PFC	16496	810	71	157
RISC-V[2]	Exception	28377	1430	170	164
RISC-V[3]	MAC	39487	2271	171	164
RISC-V[4]	Event Counters	16391	844	70	157
RISC-V[5]	CRC, PFC, MAC, Event Counters, Exception	42121	2403	170	165

the opportunity to add extensions and customizations. Nevertheless, it does not include a timing analysis engine. Yosys uses ABC [117] for logic minimization and technology mapping. The resulting gate-level netlist is the input to the OpenROAD toolchain. While commercial tools are already installed, parallelized, and ready to use within an industrial flow, OpenROAD and its flow controller are set within our multi-user compute farm [186].

B.2.4.3. Process Design Kits (PDKs)

A PDK is a set of data files describing the fabrication process of a chip for the design tools. A PDK, developed by a foundry, contains technology details, design rules, standard cell libraries, layout information, and SPICE models [24]. This paper uses two PDKs: the open-source SkyWater 130 nm and a proprietary 40 nm. Table B.1 compares relevant aspects of both technologies. Commercial tools convert the open-source PDK to a format compatible with the commercial synthesis tool.

B.2.4.4. Use Case: RISC-V

Using MetaRISC [53], five RISC-V implementations are generated using different extensions, as listed in Table B.2. Each design consists of a 32-bit 5-stage pipeline without privileged mode with support for compressed instructions and multiplication without division (RV32IMCX).

The RISC-V[1] uses an on-chip control flow integrity unit called Program Flow Check (PFC). It computes a signature with Cyclic Redundancy Check (CRC) by taking the current instruction and the last signature as input for each clock cycle. Software can compare a pre-computed signature with this run-time to ensure that the control flow is not corrupted, following the principle described in [212]. The RISC-V[2] adds an exception unit to the CPU to handle exceptions and interrupts as defined in the RISC-V privileged specifications. New signals for exception detection are added all over the pipeline and forwarded to this unit responsible for the trap handling. The RISC-V[3] incorporates a Multiply-Accumulate (MAC) unit, widely used in signal processing and machine learning applications. RISC-V[4] includes Event Counters for hardware monitoring, i.e., one cycle and one instruction counter have been added. A 32-bit Control and Status Register (CSR) has been added to activate all the possible counters, and two 32-bit CSRs represent each counter. RISC-V[5] includes all the features mentioned above.

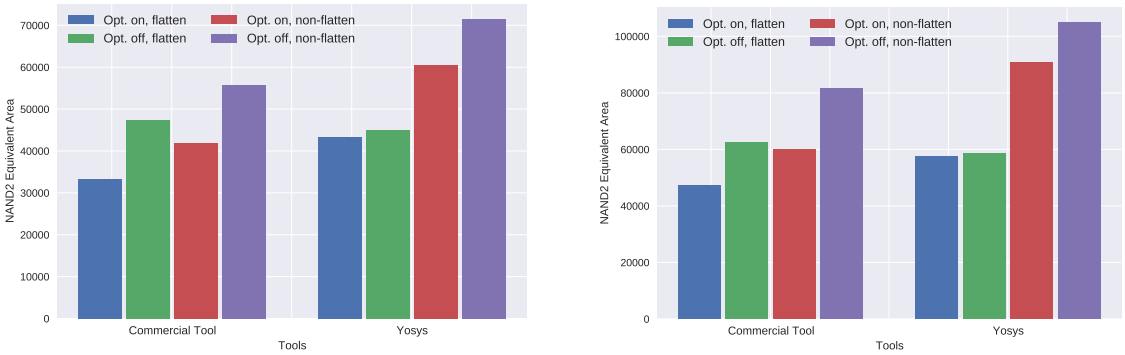
B.2.5. Synthesizing RISC-V Designs - Logic Synthesis

The design flow in Figure B.1 has been applied to generate and synthesize the RISC-V implementation listed in Table B.2. In the following, results over the different circuits are presented. The clock period is fixed to 25 ns, the maximum latency is 1 ns, and the minimum is 0.1 ns. The setup skew is maintained at 0.5 ns, the setup transition at 0.2 ns, and the hold transition at 0.1 ns.

For logic synthesis, we compare flattened and hierarchical synthesis. Moreover, Yosys offers two configurations: *ABC_AREA* and *ABC_SPEED*, for reducing area and delay, respectively. In the case of the commercial tool, synthesis is run without and with optimizations.

B.2.5.1. Comparing Area

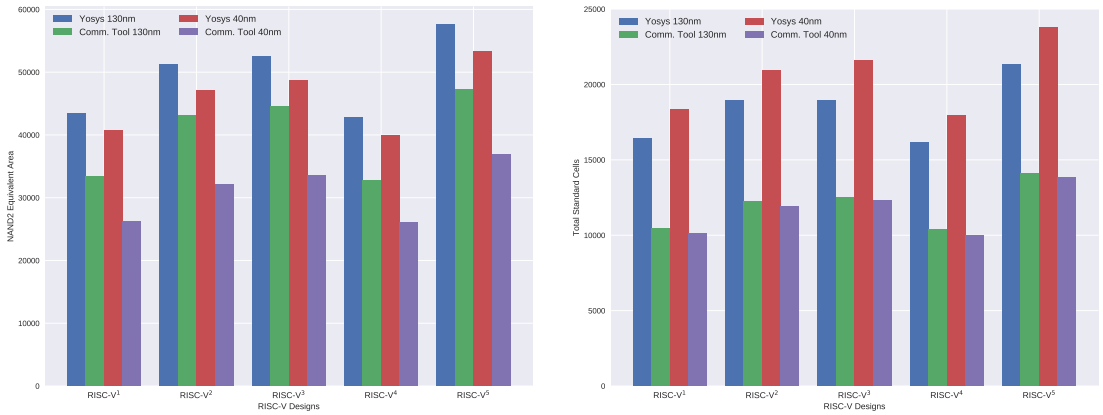
To compare the total area among two PDKs, we employ the NAND2 equivalent area, using the area of the smallest NAND cell of each technology with two inputs and one output. Figure B.2 shows how the different configurations influence the results for the RISC-V[1] and the RISC-V[5] designs. For simplicity, Figure B.2 shows results when using the open-source PDK. Similar results are obtained using the 40 nm technology: The lowest area is captured when enabling optimizations (ABC_SPEED target in Yosys) for flattened synthesis. For non-flattened designs, the commercial tool outperforms Yosys with and without optimizations.



(a) RISC-V[1] (b) RISC-V[5]

Figure B.2.: NAND2 equivalent area after logic synthesis with different configurations

All designs are synthesized using the best configurations, i.e., flattened synthesis and enabled optimizations. Figure B.3a shows that the total area when using Yosys is, on average, 1.24× and 1.49× higher than the commercial tool for the 130 nm and 40 nm technologies, respectively. Figure B.3b shows that the commercial tool uses lesser number of standard cells independently of the technology. On average, Yosys needs 1.54× more standard cells for the 130 nm technology and 1.77× for the 40 nm technology w.r.t. the commercial tool.



(a) NAND2 equivalent area (b) Total standard cells

Figure B.3.: NAND2 equivalent area and total standard cells after logic synthesis with flattened and optimized setup

B.2.5.2. Comparing WNS

The WNS and the TNS reported after logic synthesis by the commercial tool are zero for all the cases, indicating that the logic mapping is timing-driven. The reported WNS by OpenSTA after Yosys is also zero, but only when performing non-flattened synthesis. For flattened synthesis, the WNS is negative, as shown in Figure B.4, for all RISC-Vs and both PDKs. The reported slack worsens with the complexity of the designs.

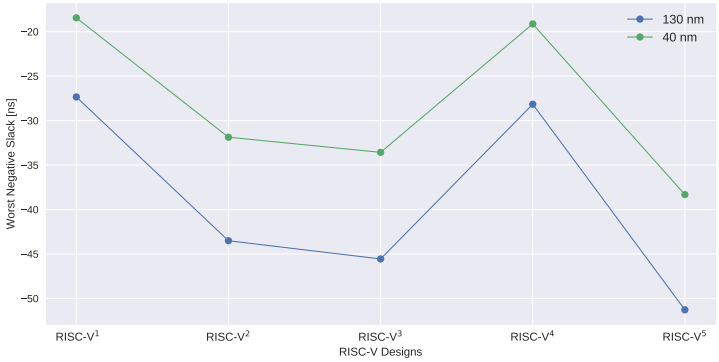
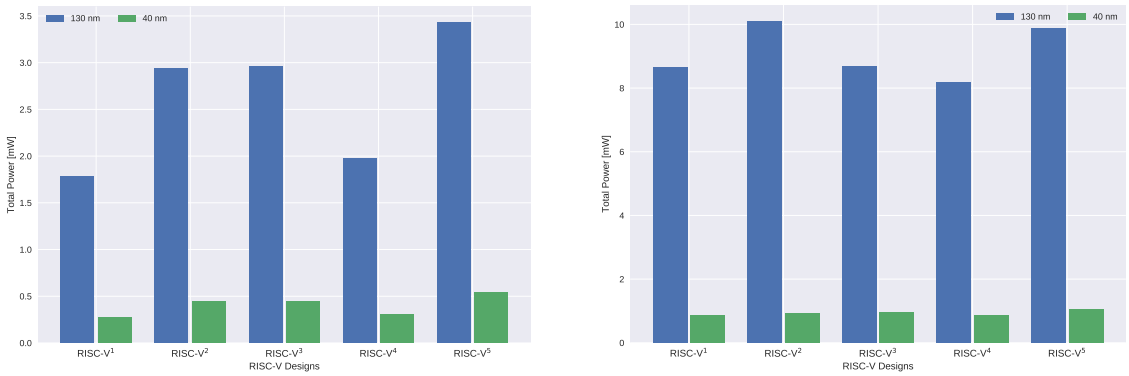


Figure B.4.: WNS per design after flattened logic synthesis using Yosys

B.2.5.3. Comparing Total Power

The reported total power after using the commercial tool is, on average, 3.81× and 2.39× lower than the value reported by OpenROAD for the 130 nm and 40 nm technology, respectively. Best results are obtained using flattened synthesis, ABC_AREA, and enabled optimizations. Figure B.5 shows that the maximum power consumed is around 3.5 mW for the commercial tool and approximately 10.1 mW for Yosys when using the 130 nm technology.



(a) Commercial tool (b) Yosys

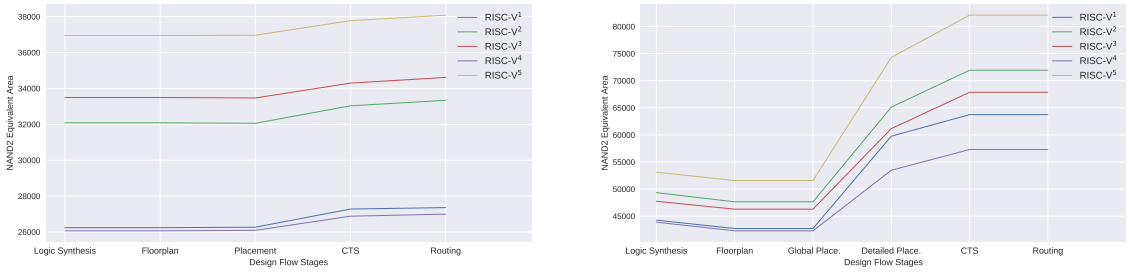
Figure B.5.: Total power after logic synthesis using the best configurations for both tools

B.2.6. Synthesizing RISC-V Designs - Physical Synthesis

The gate-level netlist obtained after logic synthesis is the input to a commercial tool and OpenROAD. We compare their results for the 40 nm technology and all RISC-Vs designs. For simplicity, the best configurations are used, i.e., flattened synthesis, ABC_AREA, and enabled optimizations.

B.2.6.1. Comparing Area

Figure B.6 shows the total NAND2 equivalent area per stage. Results show that the commercial tool results after logic synthesis are already optimized for floor planning and placement so that the total area after those stages does not increase. On average, the resulting netlist from the commercial tool occupies a lower area by a factor of 2.14× and a lesser number of standard cells by a factor of 2.41× w.r.t. the OpenROAD netlist.

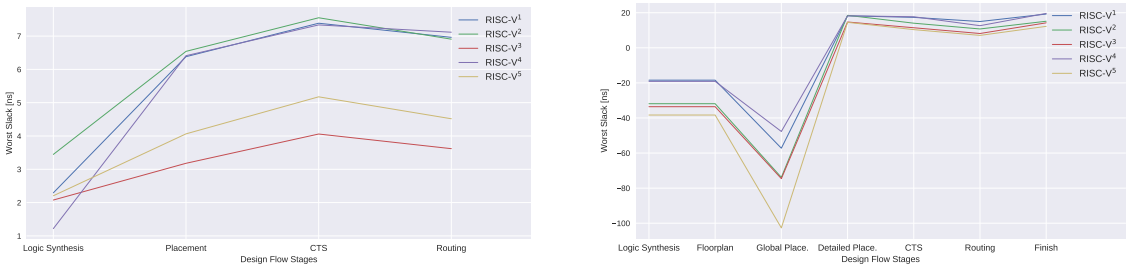


(a) Commercial tool (b) OpenROAD

Figure B.6.: Post-route NAND2 equivalent area per each stage using both tools

B.2.6.2. Comparing WNS

After routing, the WNS and TNS are zero for both tools when running flattened synthesis with enabled optimizations. Figure B.7 shows the critical path positive slack during different stages. For all designs, the average post-routing critical path in OpenROAD has a slack 2.87× higher than the reported by the commercial tool.

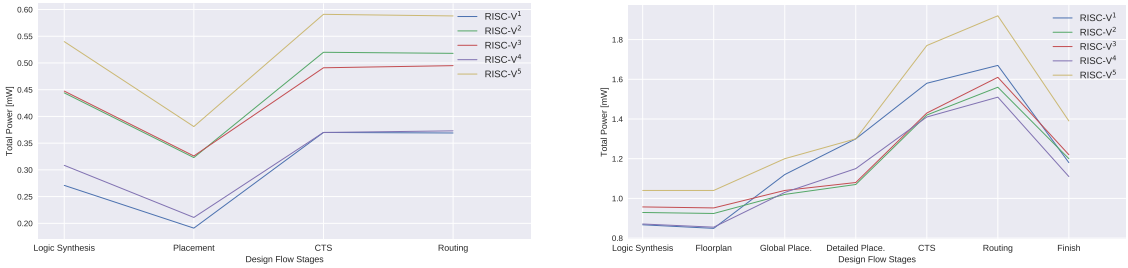


(a) Commercial tool (b) OpenROAD

Figure B.7.: Post-route critical path slack per each stage using both tools

B.2.6.3. Comparing Total Power

Figure B.8 shows the variation of the total power consumption along the design flow. OpenROAD post-routing layout consumes more power by a factor of 2.66× w.r.t. the reported values from the commercial tool.



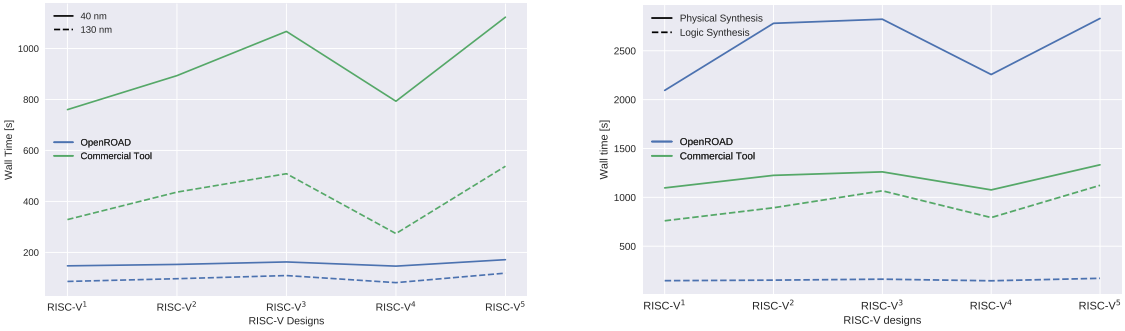
(a) Commercial Tool (b) OpenROAD

Figure B.8.: Post-route total power consumption per each stage using both tools

B.2.6.4. Runtimes

For our experiments, both tools are sequentially run on a Linux CPU Intel® Xeon® Gold 6248R at 3.00 GHz and 80 GiB system memory. For fairness in the comparison, the multi-threading options are disabled for both tools. In other cases, the commercial tool is already wrapped up and sped up so that each stage is executed in a different machine of the compute farm, based on memory and speed requirements. OpenROAD also provides the option to use multiple cores to improve runtime.

Figure B.9 shows the wall times for logic and physical synthesis. On average, Yosys (OpenROAD) is 5.91× and 4.18× faster than the commercial tool for the 130 nm and 40 nm technologies. The runtime during logic synthesis increases with the complexity of the technology used, as shown in Figure B.9a. Figure B.9b shows the wall time for floor planning until routing. On average, the physical synthesis stages consume 2.13× more time in OpenROAD than in the commercial tool. In general, the commercial tool performs logic and physical synthesis faster.



(a) Logic synthesis (b) Routing

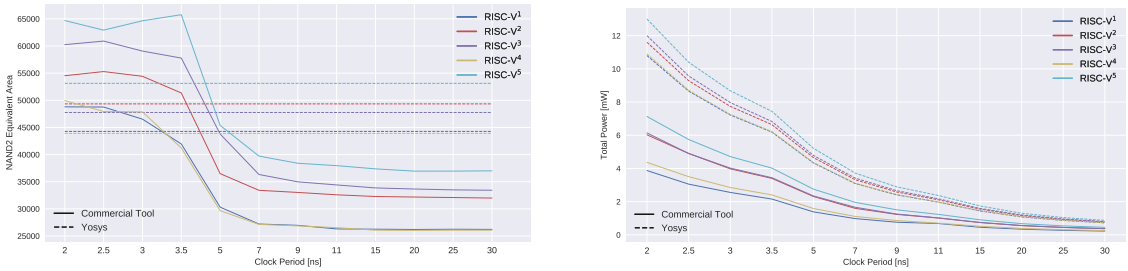
Figure B.9.: Wall time after logic synthesis and routing

B.2.7. PPA Analysis for Different Clocks

The above experiments fixed the clock to 25 ns to avoid post-routing timing violations. Figure B.10 and Figure B.11a show the PPA results after logic synthesis when sweeping the clock period. In the commercial tool, the total area is reduced when increasing the clock period until a threshold. Yosys does not consider timing constraints, and its total area across different clock periods stays constant. The commercial tool achieves a total power lower than 2 mW from a clock period equal to 7 ns. For OpenROAD, this happens for clock periods higher than 15 ns. Using a 3.5 ns clock period, the commercial tool reaches a zero worst slack for all RISC-Vs. This is never reached after running Yosys. Results after detailed placement show similar trends w.r.t. area and power. Regarding the worst slack, the netlists have already been repaired, and no timing violations appear for a clock period higher than 10 ns with OpenROAD and 5 ns with the commercial tool, as shown in Figure B.11.

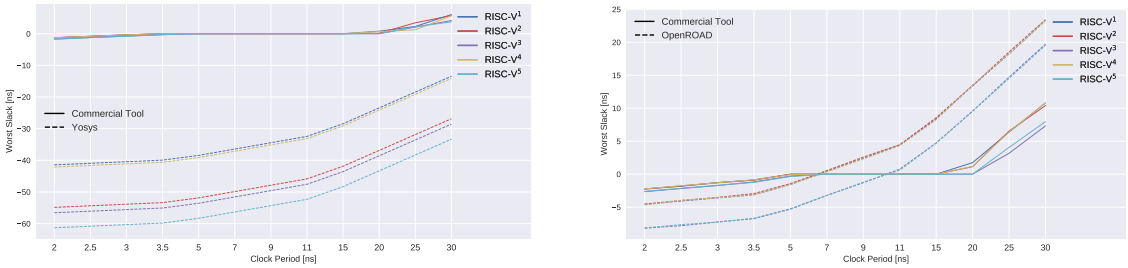
B.2.8. Summary

We outline our industrial flow from initial specifications to GDS using different RISC-Vs as use cases. We compare OpenROAD performance with a commercial EDA tool. Averaging the reported post-routing factors comparing PPA results for a 25 ns clock period, the commercial tool outperforms OpenROAD by a factor of 2.5. The commercial tool is, on average, 588 s faster without any parallelization, and it could meet the timing constraints for lower clock periods than OpenROAD. We expect this to change in the future based on the rapid development of OpenROAD and its community commitment manifested as more than 140 citations to



(a) NAND2 equivalent area (b) Total power

Figure B.10.: Total power and area after logic synthesis with different clock periods



(a) Logic synthesis (b) Routing

Figure B.11.: Worst slack after logic synthesis and routing with different clock periods

OpenROAD publications [213], 1490 commits in 2022, and 19 active pull requests [146]. In contrast to commercial tools, open-source EDA tools open the door to academic research, benchmarking, ML techniques, and the transfer of research to real-world scenarios. It is planned to include LSOracle and AutoTuner in our analysis and to use the outcomes of OpenROAD inside commercial tools and vice versa. As mentioned in [214], OpenROAD flow controllers still do not ease portability among OpenROAD and commercial tools. We expect this to change, as this would allow industries and researchers to benefit from both worlds.