

Politecnico di Torino

Politecnico di Torino

Master's Degree in Nanotechnologies for the ICTs
Micro and Nano Technologies for Integrated Systems
Academic year 2025/2026
Graduation session: September 2026

Enhancing Hierarchical ASIC Flow Automation for Heterogeneous SoC Design: The ESP Case

Supervisors:

Prof. Luca Carloni
Dr. Maico Cassel Dos Santos
Dr. Biruk Seyoum
Columbia University

Prof. Mario Casu
Politecnico di Torino

Candidate:

Leonardo Boscolo
Matricola: S338937

Abstract

Heterogeneous systems-on-chips (SoCs) combine processors, accelerators, memories, and interfaces whose implementation requirements can differ widely. As these systems grow, their physical implementation must handle not only a larger design, but also more configurations, reusable components, and target technologies. Hierarchical application-specific integrated circuit (ASIC) methodologies address this complexity by dividing the system into smaller blocks and automating their implementation. Hierarchy alone, however, does not make a flow scalable. Its automation must follow changes in the hardware configuration, distinguish reusable implementation stages from design-specific intent, and keep technology data separate from the flow logic.

This thesis investigates these requirements through the ASIC flow of the Embedded Scalable Platform (ESP), a tiled architecture and design methodology for heterogeneous systems. Configuration-aware automation was introduced for project preparation, timing constraints, and process–voltage–temperature corner selection, together with a multi-level strategy for validating changes at an appropriate implementation depth. The resulting flow was evaluated through logic synthesis for selected ESP configurations. To study reuse beyond the predefined system hierarchy, platform-specific dependencies were separated from the main implementation stages and a minimum interface for standalone designs was identified. The approach was exercised on a network-on-chip (NoC) block from logic synthesis through physical design and preliminary power estimation, without claiming signoff verification. Technology integration was addressed through a common declarative interface that translates a local technology manifest into the database used by the existing synthesis scripts. This mechanism was applied to a 28 nm technology in the Fermilab collaboration for preliminary ESP tile-area estimation. Together, these contributions show how clearer boundaries among design inputs, implementation automation, and technology configuration can improve the robustness and reuse of a hierarchical ASIC methodology. Although the evaluation is limited to the reported ESP configurations and case studies, the same principles can guide the organization of other hierarchical ASIC flows.

Acknowledgements

First, I would like to thank my academic advisor Luca Carloni, whose collaboration made it possible for me to conduct this research period for my thesis at Columbia University in the City of New York. In particular, I am grateful to have had this opportunity and for his feedback, especially for helping me refine the direction of the thesis. Second, I would like to thank my other advisor, Mario Casu, for his remarkable availability and responsiveness throughout the entire research period.

Then I would like to express my gratitude to all my supervisors, Maico Cassel Dos Santos and Biruk Seyoum. Maico guided me throughout the whole period, and he was fundamental for my academic growth. I am grateful to Biruk for challenging my critical thinking and for his immense availability.

A special thanks to the Columbia System-Level Design (SLD) group for being real friends and open to discussing new ideas, projects, proposals, especially Gabriele, William, Eugenio, Guy and Kevin. Additionally, I thank Francesca for sharing the visiting student experience with me and for every moment we spent together.

I cannot forget to thank my family for the immense support and for believing in me; lastly, I thank my friends for their companionship throughout these years.

Contents

List of Acronyms **ix**

1 Introduction **1**

1.1 Context 2

1.2 Scope 3

1.3 Motivation 4

1.4 Objectives 5

1.5 Thesis Contributions 6

1.6 Thesis Organization 7

2 Background and State of the Art **9**

2.1 ASIC Digital Design Flow 10

2.1.1 Logic Synthesis 11

2.1.2 Physical Design 14

2.2 Embedded Scalable Platform (ESP) 18

2.3 ESP ASIC Flow 21

2.3.1 Tile-based hierarchical flow 23

2.3.2 Project structure 25

2.3.3 Features of the methodology 26

3 Robustness and Configurability of the ESP ASIC Flow 28

3.1 Multi-Level Validation of the ESP Design Flow Across Releases 29

3.1.1 Validation Depth 29

3.1.2 Release-Dependent Failures Observed in Practice 30

3.2 ESP SoC Configurations 31

3.3 Flow Scalability Barriers 34

3.4 ASIC Integration Refactoring 35

3.4.1 Project and Component Preparation 35

3.4.2 SoC Generation 36

3.4.3 RTL Simulation 36

3.5 Python Refactoring 37

3.6 Conditional SDC Constraint Generation 39

3.7 Configurable PVT Corners Selection 41

3.8 Regression Tests and Validation across Different Configurations 43

4 Flow Modularization and Reuse for Standalone Designs 46

4.1 Motivation and Scope 47

4.2 ESP Design Flow Dependencies 48

4.3 Requirements for Standalone Design Reuse 51

4.4 Extraction and Configuration of the Simplified Flow 53

4.5 Standalone NoC Router Case Study 56

4.5.1 Router Configuration and Experimental Setup 56

4.5.2 Timing Constraints and Logic-Synthesis Results 57

4.5.3 Floorplanning, Physical Implementation, and Results 58

4.6 Discussion and Limitations 61

5 Technology Abstraction for Portable Logic Synthesis 62

5.1 Motivation and Scope 63

5.2 Limitations of Existing Technology Integration 63

5.3 ESP Technology Configuration Repository 65

5.4 Pre-Synthesis Integration of Ibex 67

5.5 Fermilab SoC Case Study 69

5.5.1 SoC Configuration and Synthesis Setup 69

5.5.2 Memory Integration and Validation 70

5.5.3 Synthesis Results and Additional Area Estimates 71

5.6 Discussion and Limitations 72

6 Conclusions 74

6.1 Future Work 76

6.2 Use of Artificial Intelligence in the Thesis 77

References 80

List of Figures

1.1 Historical microprocessor trends in transistor count, single-thread performance, clock frequency, typical power, and logical-core count. Reproduced from Rupp [6]. 2

2.1 Simplified view of a conventional digital ASIC design flow and its principal verification checkpoints. 10

2.2 Generalized logic-synthesis flow from design and technology inputs to a technology-mapped gate-level netlist and its associated handoff data. The enclosing *compile* block groups the main synthesis and optimization stages. 12

2.3 Hierarchical composition of an analysis view in an MMMC configuration. 13

2.4 Generalized physical design flow from the imported logical, library, technology, and constraint data to the placed-and-routed layout and its implementation views. 15

2.5 ESP tile architecture overview. Reproduced from Cassel Dos Santos et al. [24]. 19

2.6 ESP design platform flows overview. Reproduced from Mantovani et al. [11]. 20

2.7 Methodology proposed for agile ASIC development. Reproduced from Cassel Dos Santos [13]. 23

2.8 Database directory structure for the ESP ASIC methodology. 25

3.1 Main configuration files and setup steps of the ESP ASIC flow. Red boxes show individual settings, while white boxes show expandable groups of settings. 32

3.2 Generation of a `create_clock` constraint with the `TclWriter` class. Matching colors connect each part of the command to the method that produces it. 39

4.1 Innovus physical design views of the 256-bit multicast router. In a, plane 1 is green, plane 2 is blue, plane 3 is yellow, plane 4 is red, plane 5 is magenta, and plane 6 is cyan. 59

5.1 Position of the ESP Tech Config repository within a technology directory. The repository occupies the `config` branch, while the ESP wrappers and linked technology views remain separate. 65

5.2 Conceptual workflow of the ESP Tech Config repository. The generator unrolls the local technology manifest into the six TCL files shown in yellow, which are then sourced by the existing EDA flow. 66

List of Tables

3.1 Cumulative validation levels for the embedded scalable platform (ESP) design flow. 30

3.2 Effect of the clock strategy on the generated tile and network-on-chip (NoC) constraints. 40

3.3 Setup and hold views produced for representative corner selections. . . 43

3.4 Clock periods used for the Chapter 3 synthesis tests. 44

3.5 DCO simulation tests. 44

3.6 Level 1 validation of the conditional SDC constraints. 45

3.7 Synthesis results for the selected PVT corner sets. 45

4.1 Design assumptions of the hierarchical flow and their effect on the standalone router. 49

4.2 Proposed inputs for the reuse of the ASIC flow with a standalone design. 52

4.3 Pin distribution and minimum side lengths reported by the standalone pin generator. 58

4.4 Post-synthesis and post-route results for the 256-bit multicast router configuration. 60

5.1 Clock periods used for the Fermilab SoC synthesis runs. 70

5.2 Logic-synthesis area and setup-timing results for the Fermilab SoC partitions. 71

5.3 Assumption-based synthesis power estimates at `SS_corner_1`. 72

List of Acronyms

AOCV advanced on-chip variation

ASIC application-specific integrated circuit

ATPG automatic test-pattern generation

BIST built-in self-test

CDC clock domain crossing

CMOS complementary metal–oxide–semiconductor

CPU central processing unit

CSR control and status register

CTS clock-tree synthesis

DCO digitally controlled oscillator

DDR double data rate

DEF Design Exchange Format

DFM design for manufacturability

DFT design for testability

DMA direct memory access

DRAM dynamic random-access memory

DRC design-rule checking

DSA domain specific acceleration

DTCO design technology co-optimization

DVFS dynamic voltage and frequency scaling

EDA electronic design automation

EMIR electromigration and voltage drop

ESD electrostatic discharge

ESP embedded scalable platform

ETM extracted timing model

FinFET fin field-effect transistor

FPGA field programmable gate array

FSM finite-state machine

GDSII graphic data system II

GPU graphic processing unit

HDL hardware description language

HLS high-level synthesis

ILM interface logic model

ILP instruction level parallelism

IP intellectual property

LEF Library Exchange Format

LVS layout-versus-schematic

MMMC multi-mode multi-corner

MOSFET metal–oxide–semiconductor field-effect transistor

MSV multi-supply voltage

NDM New Data Model

NoC network-on-chip

OASIS Open Artwork System Interchange Standard

OCV on-chip variation

PDK process design kit

PDN power-delivery network

PLM private local memory

PnR place-and-route

PSO power shut-off

PVT process, voltage, and temperature

RC resistance–capacitance

RTL register transfer level

SDC Synopsys design constraint

SDF Standard Delay Format

SLD system-level design

SLM shared local memory

SoC system-on-chip

SOCV statistical on-chip variation

SPEF Standard Parasitic Exchange Format

SRAM static random-access memory

STA static-timing analysis

TCL tool command language

TNS total negative slack

UPF Unified Power Format

Chapter 1

Introduction

For several decades, computing advanced through a relatively stable scaling model. Moore's law described the continued growth in the number of components that could be integrated economically on a chip. Dennard scaling explained how smaller metal–oxide–semiconductor field-effect transistors (MOSFETs) could operate at proportionally lower supply voltages [1, 2]. Together, these trends allowed each generation of planar complementary metal–oxide–semiconductor (CMOS) technology to provide more transistors and higher operating speeds while keeping power density approximately constant. Improvements in semiconductor technology could therefore translate predictably into greater computing capability.

Designers used these gains to build wider datapaths, faster processors, larger memory systems, and more integrated chips. As designs grew in complexity, standard-cell methodologies and electronic design automation (EDA) became essential for turning register transfer level (RTL) descriptions into manufacturable implementations. Technology scaling, architectural development, and design automation supported one another: each generation supplied more resources, architects used them to add performance and functionality, and implementation tools helped manage the resulting complexity.

This progress depended on several benefits of scaling arriving together. Over time, however, voltage and power stopped scaling at the same rate. Integrating more transistors no longer guaranteed similar gains in general-purpose performance or energy efficiency [3, 4]. Further progress therefore required changes in devices, the interaction between technology and design, computer architecture, and system implementation. Section 1.1 describes these challenges and the resulting move toward specialized, heterogeneous systems.

1.1 Context

Strained channels, high-permittivity gate dielectrics with metal gates, stricter patterning rules, and non-planar devices such as fin field-effect transistors (FinFETs) allowed transistor density to keep increasing. They did not, however, restore proportional supply-voltage scaling [3]. The relationship between technology and design changed as well. Beginning around the 90-nm node, design for manufacturability (DFM) mainly passed yield-related restrictions from foundries to designers through stricter design rules. Design technology co-optimization (DTCO) replaced this largely one-way exchange with earlier and more frequent collaboration. Foundries, library developers, EDA providers, and product designers could then use design results to guide technology choices before the process was fixed [5].

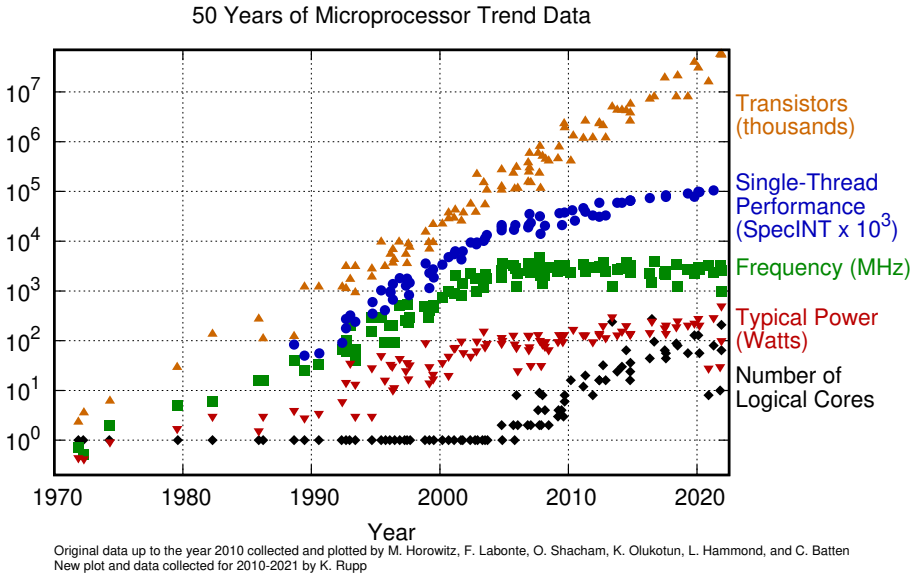


Figure 1.1: Historical microprocessor trends in transistor count, single-thread performance, clock frequency, typical power, and logical-core count. Reproduced from Rupp [6].

At the architectural level, the loss of proportional voltage scaling led to the power wall shown in figure 1.1. Raising clock frequencies or using more complex cores to exploit instruction level parallelism (ILP) could no longer sustain earlier performance gains within practical power budgets. Homogeneous multicore processors moved the focus from single-thread performance to parallel throughput. Their gains were still limited by the parallelism of applications, memory bottlenecks, and chip-level power constraints. In the resulting *dark silicon* regime, an increasing fraction of the available transistors cannot operate at peak performance at the same time [7]. General-purpose processors remain essential, but homogeneous architectures no longer turn each technology generation into comparable gains in system performance and energy efficiency [4].

At the same time, higher levels of integration made the system-on-chip (SoC) an important unit of design. This reuse-oriented approach combines processing, memory, and interface cores on one die, increasing system integration while keeping design costs under control [8].

Within this setting, domain specific acceleration (DSA) offers another path to higher efficiency. Datapaths, numerical representations, memory organization, and parallelism can be adapted to a class of workloads, reducing the instruction-processing and data-movement overheads of general-purpose processors [9]. Since specialized hardware serves only its target workloads, modern SoCs combine programmable processors with graphic processing unit (GPU) and hardware accelerators. Software can then select the resource suited to each part of an application. Heterogeneous SoCs preserve system-level flexibility while improving efficiency in selected domains.

Heterogeneity moves much of the engineering difficulty from the individual components to their integration. Components written in different languages and built around different interfaces and assumptions must share communication and memory resources. They must also provide consistent control and data-transfer mechanisms to software and meet the same verification, power, timing, and physical design requirements. As the variety of components grows, case-by-case integration becomes difficult to maintain. Reusable interfaces, regular communication structures, and hardware software co-design are therefore needed. Even with a reusable architecture, implementation remains difficult when the flow is tied to one system configuration, semiconductor technology, or EDA environment.

The embedded scalable platform (ESP) addresses this problem through a modular, tile-based, network-on-chip architecture and a related system-level design methodology [10, 11]. Processors, accelerators, memories, and I/Os communicate through regular interfaces. Accelerator sockets and automated flows support components developed through different design approaches, including high-level synthesis (HLS). By combining this regular structure with specialized components, ESP provides the concrete case in which this thesis examines how architectural modularity relates to physical implementation.

1.2 Scope

This thesis covers digital-design analysis, hierarchical application-specific integrated circuit (ASIC) implementation, semiconductor-technology integration, and design-flow automation. At the RTL, the work examines design hierarchies and individual blocks, together with clock domains and clock domain crossings (CDC). This provides the

information needed to generate configuration-aware Synopsys design constraints (SDC) and describe the relationships between synchronous and asynchronous regions.

The implementation work considers both complete SoC configurations and reusable blocks, although they are not evaluated to the same depth. Changes to the complete SoC flow are assessed through the validation levels reported in Chapter 3. The standalone case study instead reaches physical implementation and preliminary power estimation. Depending on the experiment, the relevant EDA stages include logic synthesis, physical implementation, timing analysis, and power analysis.

The technology work covers the process design kits (PDKs) and implementation collateral needed to map a design to a fabrication process. This includes standard-cell libraries, memory compilers, timing and physical library views, and the metadata that describes them. The thesis covers the direct integration of this collateral and the organization of technology-specific inputs. The aim is to let the synthesis methodology support additional technologies without spreading their metadata throughout the flow scripts.

Finally, the thesis includes the scripting infrastructure that connects the design, tools, and technology inputs. The automation uses Python, Make, and Tcl, with an emphasis on parameterization, maintainability, and clear interfaces throughout the flow. It also considers how the flow can be distributed across independent repositories without requiring a single source-tree layout.

1.3 Motivation

Examined in detail in Chapter 2, ESP is an open heterogeneous SoC platform supported by several design flows. Its original hierarchical ASIC flow was developed around a specific platform release, repository organization, tool environment, and semiconductor technology. Although this combination provided a working path for complete ESP systems, these dependencies made the methodology difficult to maintain as the platform evolved and difficult to reuse in other design and technology contexts. The thesis therefore focuses on strengthening the established flow before attempting broader changes to its tool ecosystem.

Understanding and exercising the existing flow revealed limitations that could not be addressed as isolated backend issues. The flow had been validated with an earlier ESP release, but changes in the independently maintained ESP repository had invalidated assumptions in the flow repository. Configuration handling was also incomplete: several Python and tool command language (TCL) generators assumed particular clock

strategies and enabled interfaces. As a result, choices exposed by ESP did not always propagate consistently through the ASIC flow.

The hierarchical organization of the flow was valuable for implementing a complete heterogeneous SoC, yet its block structure was rigid. Small designs could not readily be extracted from the ESP hierarchy and reused as standalone implementation targets. Moreover, although the methodology was conceived as a general flow capable of supporting multiple technology nodes, its implementation was closely tied to one technology, with collateral and metadata embedded throughout the scripts. These limitations motivated both the extraction of a reusable block-level flow and a clearer interface between implementation logic and technology-specific inputs.

The resulting work treats flow robustness, design reuse, and technology abstraction as mutually dependent concerns. Configuration-aware automation is needed to keep the flow aligned with an evolving platform; a minimal design interface is needed to apply the methodology outside the complete ESP hierarchy; and centralized technology metadata is needed to integrate new process collateral without duplicating technology assumptions throughout the scripts. Together, these concerns define a practical foundation on which later work can investigate additional features and alternative tool ecosystems.

1.4 Objectives

The objective of this thesis is to establish a more reliable and extensible foundation for the ESP hierarchical ASIC flow. The work addresses the obstacles encountered while maintaining the established flow, adapting it to simpler and reusable designs, and integrating additional technologies. This objective is organized around three research questions:

RQ1: How can the robustness and configurability of the ESP ASIC implementation flow be improved across selected platform releases, SoC configurations, configuration-dependent timing constraints, and process, voltage, and temperature (PVT) corners?

RQ2: To what extent can the ESP ASIC implementation flow be decoupled from platform-specific dependencies and reused for standalone RTL designs?

RQ3: How can a common configuration interface represent the technology-dependent collateral references and metadata required to integrate additional semiconductor technologies into the ESP logic-synthesis flow?

The first question addresses release maintenance, configuration propagation, conditional timing constraints, and selected PVT corners. Its conclusions are limited to the configurations and validation levels reported in this thesis. The second examines how platform-specific assumptions can be isolated so that the implementation infrastructure can also serve standalone RTL designs. The third examines how a fixed configuration structure can be reused across technologies while each local configuration supplies the collateral references and metadata of its target technology. Together, the questions progress from strengthening the established flow to increasing its independence from a particular design hierarchy and technology configuration.

1.5 Thesis Contributions

This thesis makes three principal contributions, corresponding to the research questions introduced above:

1. The ESP ASIC flow was updated for release 2026.1.0 and made more robust across selected SoC configurations. The work corrected failures in the public ASIC preparation procedure, refactored the Python and TCL generation code, and introduced conditional tile and standalone network-on-chip (NoC) timing constraints for the three clock strategies and optional interfaces. A multi-level validation framework was defined, and selected configurations were tested through Level 1. Together, these changes make configuration choices propagate more consistently through the flow and provide a clearer basis for maintaining it as ESP evolves.

2. A proof of concept demonstrated that the implementation methodology can be reused for a small standalone design block outside the complete ESP hierarchy. A flow closely derived from the hierarchical ASIC flow was exercised from logic synthesis through physical design, including Innovus implementation checks, parasitic extraction, and preliminary power estimation. This experiment demonstrates block-level reuse while deliberately stopping short of signoff verification: Calibre design-rule checking and layout-versus-schematic verification were not performed.

3. Technology-dependent database entries were organized through a fixed, technology-independent YAML template. For each target technology, the template is instantiated as a local `tech.yaml` manifest and populated with the corresponding collateral references and parameter values. The version-controlled template, Make targets, and Python generator provide a common configuration interface, while the populated manifest remains untracked to keep installation-specific and sensitive values outside

the repository. The generator translates the manifest into the Tcl database consumed by the ESP flow. This mechanism is applied in the Fermilab collaboration to configure a 28 nm technology for preliminary ESP tile-area estimation. Chapter 5 reports the extent of the logic-synthesis evaluation, while physical implementation and signoff remain outside the scope of this work.

Collectively, these contributions establish a more robust, reusable, and technology-aware foundation for the ESP ASIC methodology. They separate platform, design, and technology concerns while documenting the validation boundaries of each implementation and case study. This foundation can support later investigation of additional technologies and alternative EDA ecosystems, but those extensions are outside the present scope.

1.6 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 establishes the technical background required by the subsequent chapters. It reviews the digital ASIC design flow, introduces the ESP architecture, and describes its hierarchical implementation methodology and the associated technology inputs.

Chapter 3 addresses RQ1. It first defines the validation levels used in the thesis and describes the configuration space of the ESP SoC and ASIC flow. It then presents the corrections required by the public ASIC preparation procedure, the Python refactoring, and the conditional generation of tile and NoC timing constraints. The chapter next discusses configurable PVT corners and concludes with regression tests across selected configurations.

Chapter 4 addresses RQ2 by identifying the dependencies that tied the implementation flow to the complete ESP hierarchy and by defining the minimum interface required for a standalone design. It then describes the extraction and configuration of a simplified flow and evaluates its reuse through a NoC-router case study carried from RTL synthesis through physical design. The chapter concludes by delimiting the demonstrated portability and the remaining restrictions.

Chapter 5 addresses RQ3 by presenting a common configuration interface for the technology-dependent inputs required by logic synthesis. A fixed YAML template defines the database structure, while a local manifest supplies the collateral references and parameter values for each target technology. Make-based automation and a Python generator translate the manifest into the Tcl database consumed by the ESP flow. The mechanism is examined through the Fermilab case study on preliminary area estimation

for ESP tiles, after which the chapter states the validation boundary and the work required to extend the approach to the backend flow.

Finally, Chapter 6 summarizes the findings in relation to the three research questions, states the overall conclusions, and identifies directions for future work toward broader design, technology, and tool portability.

Chapter 2

Background and State of the Art

The preceding chapter argued that progress toward an open implementation path for ESP depends first on making its established ASIC methodology more robust, reusable, and independent of particular technologies. Addressing the resulting research questions requires a common account of both the general implementation process and the platform-specific, technology-specific, and tool-specific assumptions embodied in that process. This chapter therefore establishes the technical foundation against which the changes and assessments in the subsequent chapters can be understood.

The selected topics correspond to the principal dependencies examined by the thesis. The digital ASIC design flow supplies the concepts needed to evaluate configurability and block-level reuse in RQ1 and RQ2. The ESP architecture and its existing hierarchical ASIC flow define the concrete system, methodology, and initial limitations to which those questions apply. Their technology inputs support RQ3 by identifying the collateral and metadata required to target a semiconductor process. The chapter's central contribution is thus a structured baseline that separates design, platform, and technology concerns while showing how they interact in an RTL-to-GDS methodology.

The chapter first reviews the stages and inputs of a conventional digital ASIC flow and then introduces ESP and its hierarchical implementation flow. This progression provides the terminology and implementation context used throughout Chapters 3 to 5.

2.1 ASIC Digital Design Flow

A digital ASIC design flow transforms a design specification into a layout that can be sent to a foundry. The overview presented in this section follows the conventional flows described by Weste and Harris and Cassel dos Santos [12, Sec. 14.4][13, Ch. 2, Sec. 2.1]. As summarized in Figure 2.1, this transformation progresses through behavioral, structural, and physical representations. The forward path alone is not sufficient: the output of each main implementation stage must be verified against the representation and requirements from which it was derived. Implementation and verification therefore form interrelated parts of an iterative RTL-to-GDSII flow, and a failed check returns the design to the appropriate preceding stage. The verification methods named below are representative, commonly used checks associated with each stage rather than an exhaustive taxonomy; the figure groups them into three checkpoints to preserve a general view of the flow.

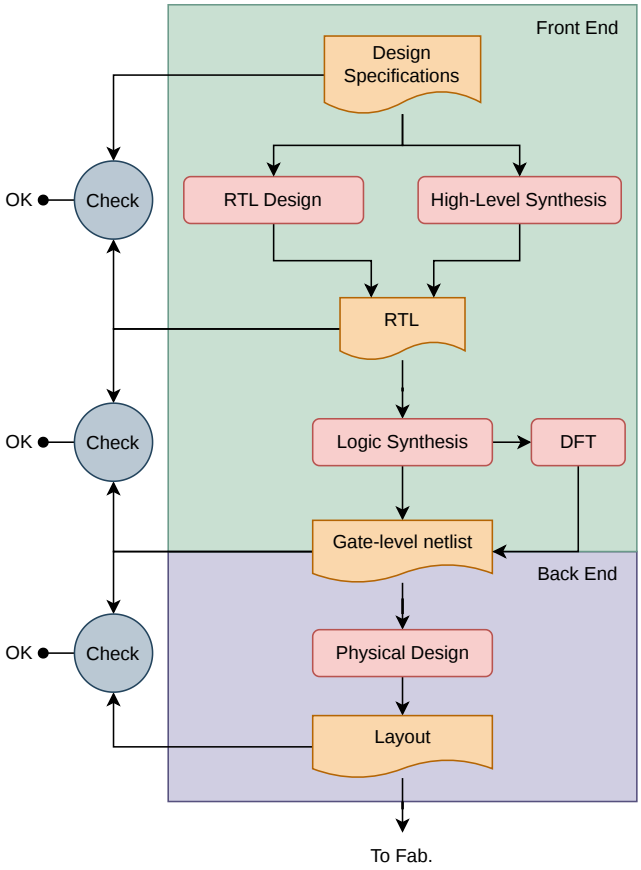


Figure 2.1: Simplified view of a conventional digital ASIC design flow and its principal verification checkpoints.

At an abstraction above RTL, the intended operation may also be captured in behavioral or functional models, such as SystemC or C++ models, without committing to RTL

implementation details. Because this thesis takes RTL as its starting point, Figure 2.1 does not represent these models separately; for simplicity, they are subsumed under *design specifications*.

The flow begins with the *design specifications*. These guide either manual *RTL design* or *HLS*, both of which produce an RTL description. At the first verification checkpoint, RTL simulation and coverage analysis are commonly used to check the RTL against the design specifications. Only an RTL representation that satisfies the required checks proceeds to the next implementation stage.

Logic synthesis then optimizes the RTL and maps it to a target standard-cell library. In the same part of the flow, *design for testability (DFT)* includes techniques such as scan insertion and built-in self-test (BIST), while automatic test-pattern generation (ATPG) produces patterns for testing the resulting structures. The principal output is a *gate-level netlist*. Representative checks of this netlist with reference to the RTL and its constraints include formal-equivalence checking, static-timing analysis (STA), gate-level simulation, and power analysis.

The gate-level netlist is subsequently converted by *physical design* into the *layout*. Verification at this final interface checks the layout against the gate-level netlist, the design constraints, and the manufacturing rules. Prominent methods include post-layout STA, design-rule checking (DRC), layout-versus-schematic (LVS), and DFM, together with parasitic extraction and power, thermal, electromigration and voltage drop (EMIR), and signal-integrity analyses. These checks may require iteration within physical design or a return to earlier stages. Only after all checks required by the applicable design methodology and fabrication process pass is the layout released *to the fab*.

2.1.1 Logic Synthesis

Logic synthesis converts the behavioral and sequential content of an RTL description into an optimized structural implementation built from cells available in the target libraries. Figure 2.2 presents a generalized sequence: its synthesis and optimization basis follows De Micheli, while the organization of modern synthesis stages and analysis views follows current industrial practice [14, Chs. 3, 7–10][15, 16]. The precise commands, order of input loading, and distribution of optimizations among the stages remain tool- and methodology-dependent.

The synthesis inputs are grouped into four classes to avoid tying the flow to a particular file organization. *Design hardware description language (HDL)* denotes the RTL sources, commonly written in SystemVerilog, Verilog, or VHDL. *Library data* describes the

available standard cells and may also cover hard macros, memories, I/O libraries, and power-management cells. It includes logical and timing views, typically in Liberty (`.lib`) format, and, when physical awareness is used, abstract physical views such as Library Exchange Format (LEF). *Technology files* collect the process-dependent information needed by the selected methodology, which can include a technology file (`.tf`) or technology LEF, layer maps, and corner data for modeling interconnect parasitics. The exact boundary between library and technology data varies across flows; in particular, physically aware synthesis consumes more physical collateral than conventional wire-load-based synthesis. Finally, *design constraints* is used here as an umbrella for timing constraints, usually expressed through SDC, and for power intent, for example in Unified Power Format (UPF). These inputs can describe clocks, interfaces, exceptions, operating modes, voltage domains, and related implementation requirements.

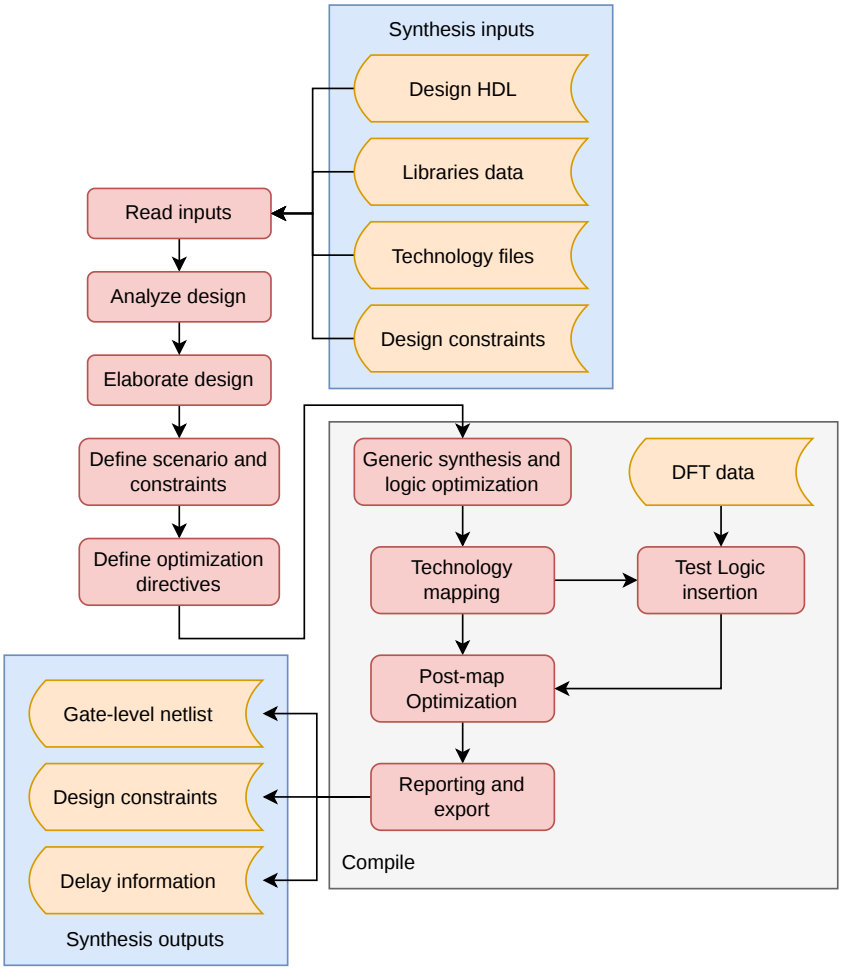


Figure 2.2: Generalized logic-synthesis flow from design and technology inputs to a technology-mapped gate-level netlist and its associated handoff data. The enclosing *compile* block groups the main synthesis and optimization stages.

The first three steps construct the synthesis tool's internal design representation. *Read inputs* denotes loading the required source, library, technology, and constraint data; it is shown once at the start even though an actual tool may load or activate individual inputs later. *Analyze design* parses the RTL, performs language and consistency checks, and records the design units in the local database. *Elaborate design* then resolves parameters and generated structures, links module instances into the active hierarchy, infers RTL operators and storage elements, and excludes unreferenced design units. Simple canonicalization and cleanup, including constant propagation and removal of unconnected logic, may also begin at this point.

Define scenario and constraints establishes the conditions under which the design is analyzed and optimized. A simple run may use one constraint set and one operating corner, whereas a multi-mode multi-corner (MMMC) setup evaluates multiple functional or test modes across several timing and interconnect corners. As illustrated in Figure 2.3, each active analysis view combines a constraint mode, generally formed from one or more SDC files, with a delay corner. A delay corner in turn combines an resistance–capacitance (RC) corner with a timing condition, which associates a library set with a particular PVT condition; variation models such as on-chip variation (OCV) may further refine timing behavior. Depending on the process and extraction methodology, RC corners may be represented by CapTable or more detailed QRC technology files. This hierarchy allows synthesis to optimize against the relevant views concurrently rather than against a single nominal condition.

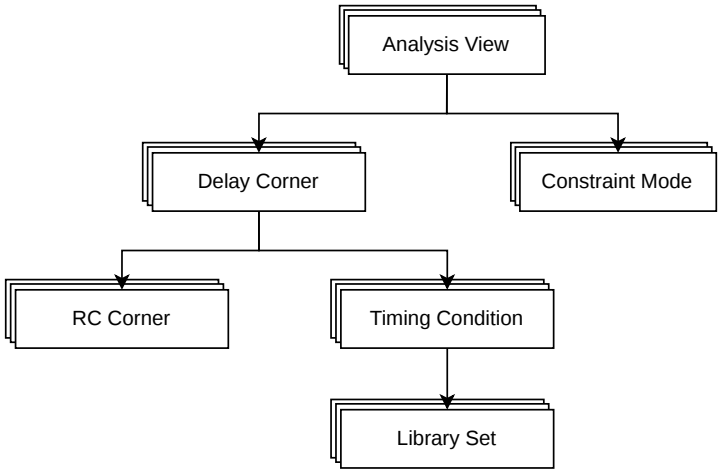


Figure 2.3: Hierarchical composition of an analysis view in an MMMC configuration.

Define optimization directives supplies objectives beyond the functional and timing constraints. Examples include the synthesis effort, cost or path groups, path adjustments, and restrictions such as cells that must not be used. Together with the active scenarios,

these directives determine how the subsequent *compile* stages trade timing, area, power, and runtime.

During *generic synthesis and logic optimization*, RTL processes, arithmetic operations, and finite-state machines (FSMs) are translated into a technology-independent network of Boolean and sequential elements. The network is then simplified and restructured through multiple-level logic transformations before committing it to specific cells. *Technology mapping*, also called *library binding*, replaces this generic network with an equivalent interconnection of cells from the enabled libraries while optimizing objectives such as area and delay. Mapping can be viewed as covering portions of the generic network with compatible library cells, although practical tools combine several algorithmic and rule-based techniques.

Test logic insertion uses the supplied DFT data, such as scan and test protocols, test constraints, and ATPG-related setup information, to add scan chains, BIST, compression logic, or other required test structures. Its exact position relative to mapping may vary. *Post-map optimization* subsequently improves the bound netlist using timing-aware incremental transformations such as cell resizing and remapping, buffering, local restructuring, and, where enabled, register retiming. Since such changes interact with test logic and with constraints, optimization is generally iterative rather than a single pass.

The final *reporting and export* step records the achieved timing, area, power, and test results and creates the artifacts required by downstream checks and physical design. As grouped in the figure, the principal outputs are a gate-level netlist, normally written in Verilog; the propagated design constraints and power intent, together with a MMMC configuration when applicable; and delay information, commonly exported as an Standard Delay Format (SDF) file for timing-accurate gate-level verification.

2.1.2 Physical Design

The frontend representation deliberately abstracts many properties that become essential when the gate-level netlist is realized in silicon. Power and ground connections are commonly implicit, the integrity of their physical distribution is unknown, cell locations and wire geometries are undefined, and high-fanout nets such as clocks are initially treated as ideal. Physical design must therefore instantiate the geometric views of cells and macros, construct and connect the power supplies and voltage domains, place and route the design, realize the clock network, and re-evaluate timing with physical interconnect effects. It also introduces non-functional physical cells, including tie cells,

power and ground pads, filler cells, decoupling-capacitor cells, end-cap cells, and well-tap cells. The overview below follows the place-and-route (PnR) flows described by Weste and Harris and Kahng et al., with variation-aware timing drawn from Bhasker and Chadha and the advanced-node refinements supported by contemporary implementation practice [12, Secs. 1.10 and 14.4.2][17, Chs. 1, 3–8][18, Ch. 10][19, 20, 21, 22, 23].

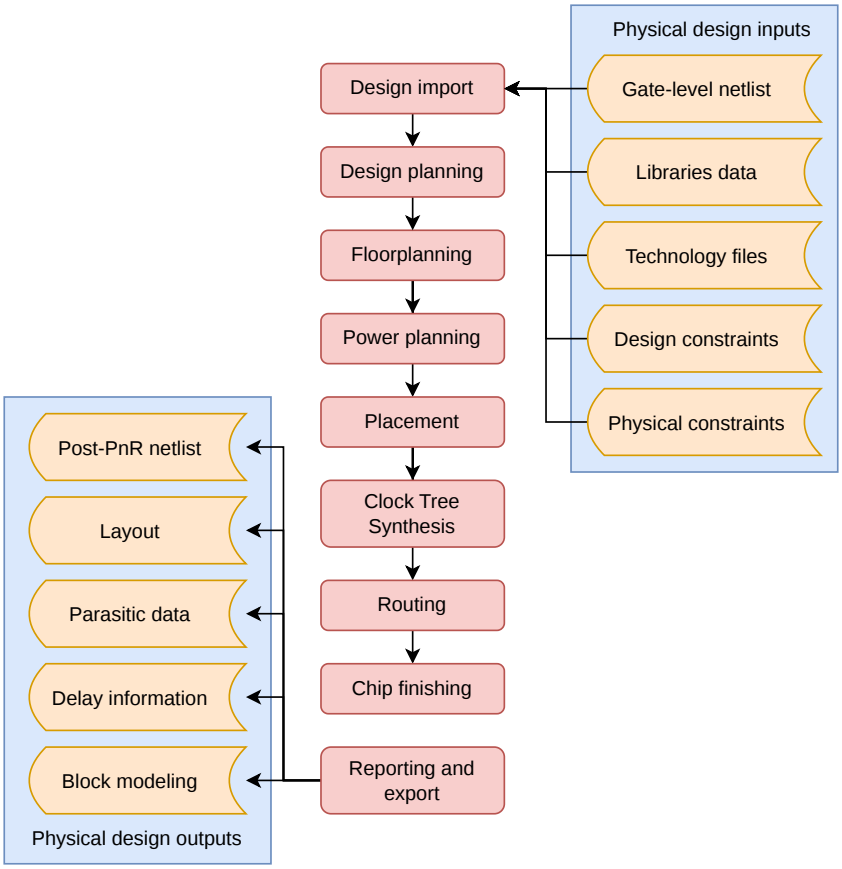


Figure 2.4: Generalized physical design flow from the imported logical, library, technology, and constraint data to the placed-and-routed layout and its implementation views.

As shown in Figure 2.4, the inputs extend those of logic synthesis with the information required to construct a legal layout. The *gate-level netlist* provides the cells and connectivity produced by synthesis. *Library data* supplies timing models and abstract physical views for standard cells, I/O, power-management libraries and hard IP macros; formats include LEF and tool-native databases such as Synopsys New Data Model (NDM). Unlike the primarily logical views used during synthesis, these views expose cell dimensions, pin shapes, routing obstructions, and other information required for placement and routing. *Technology files* describe the layer stack, routing rules and layer maps, and interconnect corners. Together with library characterization, timing may use conventional OCV or more detailed like advanced on-chip variation (AOCV)

and statistical on-chip variation (SOCV) models when supplied by the methodology. *Design constraints* include timing intent, power intent and voltage domains, and the MMMC configuration when used. Finally, *physical constraints* capture the die and core limits, pad or bump requirements, utilization targets, hierarchy and partitioning, and the number and dimensions of blocks, tiles, memories, and other macros that must fit on the device.

Design import reads these inputs, links the logical instances to their physical and timing views, and initializes the implementation database. *Design planning* denotes the engineering activity through which the designer defines how the subsequent implementation phases will be handled. This includes selecting the hierarchical or flat strategy, identifying partitions and major interfaces, and planning package connections, voltage areas, and data flow so that floorplanning, placement, and routing can converge without excessive iteration.

Floorplanning turns this high-level plan into a concrete physical framework. It selects the die and core areas, creates placement rows and grids, establishes routing rules and blockages, assigns I/O pins and bump or pad locations, and accommodates pad-level protection such as electrostatic discharge (ESD) structures. Spaces for soft blocks are reserved, while memories and other hard macros are placed with orientations and origins compatible with the placement grid. Halos, fences, and routing blockages can protect macro boundaries and preserve access to their pins. End-cap cells terminate physical boundaries in the placement grid, well-tap cells provide regular substrate and well contacts to limit latch-up risk, and decoupling cells locally support the supplies; the exact insertion point of these cells is flow-dependent. Pad count and pitch can make the design *pad-limited*, whereas the logic and macro area can make it *core-limited*. The resulting floorplan balances die area and cost against utilization, estimated wire length, congestion, timing, and routability.

Power planning constructs the global power-delivery network (PDN) that delivers power and ground to every instance and voltage domain. Power rings, stripes and rails are drawn, assigned to suitable metal layers and properly sized. Wider or denser structures reduce resistive voltage drop and current density, limiting EMIR effects, but consume more routing resources.

Placement assigns locations to standard cells within the floorplanned regions and optimizes the design under physical and electrical constraints. Global placement first seeks a good approximate distribution using objectives such as estimated wire length, congestion, density, and total negative slack (TNS); overlap may still be present at this stage. Legalization and detailed placement then remove overlap, align cells to valid

rows and sites, and refine local positions for pin access, timing, power, and routability. Incremental sizing, buffering, and cell movement may be interleaved with these steps as physical estimates become more accurate.

During *clock-tree synthesis (CTS)*, the clock is propagated from its source to the clock pins of sequential elements. Differences in arrival time produce clock skew, while the large fanout and distributed wiring create substantial capacitive and resistive loads. A buffered tree partitions this load, restores slew, and permits branch delays to be controlled through topology, buffer placement, and sizing. Conventional flows seek bounded or nearly balanced skew, whereas modern approaches can deliberately exploit useful skew. Cadence's *Clock Concurrent Optimization* (CCOpt), for example, builds the clock network while optimizing data paths with propagated clocks and variation-aware timing, improving closure across analysis views rather than treating clock and data optimization separately. By avoiding an unnecessarily zero-skew solution, such approaches can also reduce clock buffering, coincident switching, and the associated peak-current and IR-drop pressure. Reset is not a clock, but its similarly high fanout often makes it another candidate for a carefully designed buffer tree. Timing is rechecked after clock propagation because the earlier ideal-clock assumption no longer applies.

Routing connects the placed cell and macro terminals with wires and vias on the available metal layers while respecting the technology rules. Global routing divides the layout into coarse *gcells*, containing on the order of ten routing tracks per direction. These estimates expose congestion and guide placement or routing adjustments. Detailed routing then selects the exact tracks, layers, widths, spacings, and vias needed to legalize every connection, while optimizing wire length, congestion, parasitic delay, and manufacturability. Signal integrity is a another crucial routing objective: coupling between neighboring wires can alter delay or create crosstalk noise, while supply integrity depends on the interaction between switching activity and the PDN.

Finally, *chip finishing* inserts filler and other finishing cells, completes remaining physical structures, and prepares the database for signoff layout generation. In-design checks commonly include abstract-view design-rule and connectivity checks, antenna checks, and preliminary power-integrity analysis. Advanced nodes may require additional checks for pin access, dummy-gate structures, patterning, and other process-specific restrictions. These checks help remove implementation errors before full GDS-level physical verification, but do not replace signoff DRC, LVS, extraction, timing, and reliability analysis.

The concluding *reporting and export* stage packages the reports and implementation views shown in Figure 2.4. The *post-PnR netlist* reflects the transformations introduced

during physical implementation, notably the realized clock network and its buffers, and may also contain physical-only cells. Many such cells have no functional signal pins, although they may retain power, ground, or well connections. The complete mask layout is commonly exported as graphic data system II (GDSII) or as a Open Artwork System Interchange Standard (OASIS) (`.oas`), a more compact successor designed for large hierarchical layout data. For hierarchical reuse, placement and routing data may instead be written in Design Exchange Format (DEF), while a reduced physical abstraction of a block or macro may be exported in LEF. *Parasitic data* describes the resistance and capacitance of the routed interconnect, commonly through Standard Parasitic Exchange Format (SPEF), and supports later checks like accurate timing analysis, signal-integrity, and power analysis. *Delay information*, typically in SDF, enables timing back-annotation for post-PnR gate-level simulation. Hierarchical or partitioned methodologies may additionally export *block models* for top-level integration: an interface logic model (ILM) preserves the interface logic as a reduced netlist together with relevant parasitic and constraint information, whereas an extracted timing model (ETM) abstracts the block timing into a Liberty-like model. These artifacts support the final verification loop described in Figure 2.1.

2.2 Embedded Scalable Platform (ESP)

ESP is an open-source research platform for heterogeneous SoC design and programming. It has been developed over the course of more than a decade, motivated by the rise of specialized hardware accelerators, deployed for reaching energy efficiency performances. ESP's vision is a platform, which is the combination between an architecture and a methodology.

The ESP architecture is a scalable and modular heterogeneous tile grid, connected at the top-level with a 2D-mesh multi-plane NoC. Each tile is encapsulated by a socket, which consists of the collection of additional structures and modules needed for the tile to be integrated in the ESP environment. It also serves as a decoupling from the NoC and provides multiple services to the intellectual property (IP) block of each tile, including dynamic voltage and frequency scaling (DVFS), coherence and direct memory access (DMA).

There are currently five types of tiles in the ESP architecture, as shown in Figure 2.5.

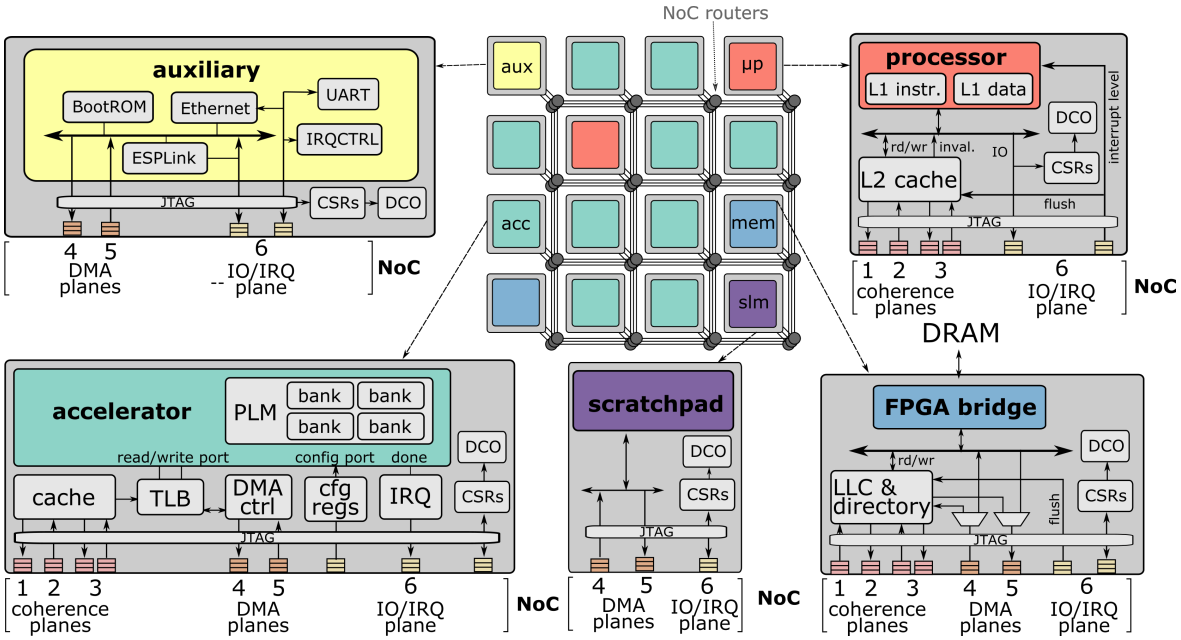


Figure 2.5: ESP tile architecture overview. Reproduced from Cassel Dos Santos et al. [24].

The *processor tile* contains a third-party developed central processing unit (CPU). The supported options include the 32-bit SPARC Leon3 core, the 32-bit RISC-V Ibex core, and the 64-bit RISC-V CVA 6 (formerly known as Ariane). Both Leon3 and CVA6 have an internal L1 cache, while Ibex only presents a buffer register. Then ESP supports the possibility to instantiate an optional ESP L2 cache, which for Ibex becomes its first level cache in practice. Additionally, CVA6 and Leon3 are capable of booting-up Linux SMP.

The *memory tile* provides a channel to external memory. Multiple memory tiles can be instantiated to increase the memory bandwidth of large SoC designs. Each memory tile serves a separate partition of the global address space, and the logic for routing requests to the correct tile is automatically generated. When the cache hierarchy is enabled, the memory tile also contains a partition of the last-level cache (LLC), which supports coherent accesses from processors and accelerators [13, 11].

The *IO tile* hosts the non-memory input/output interfaces, including Ethernet and universal UART. It also contains shared components such as the boot ROM and the interrupt controller. The Ethernet interface enables remote connections through Secure Shell (SSH) and supports the ESPLink debug application [13, 11].

The *shared local memory tile* increases the on-chip memory capacity. It contains a software-managed scratchpad outside the coherent address space, which can be shared by multiple CPUs and accelerators through DMA. This tile is especially useful for ASIC

prototypes without a double data rate (DDR) memory controller, where accesses to external memory have a higher cost [13, 24].

The *accelerator tile* contains a loosely coupled hardware accelerator. It executes a coarse-grained task after being started by a processor through a device driver. The accelerator can access the memory hierarchy with different coherence modes, from non-coherent DMA to full participation in the coherence protocol through a private L2 cache. ESP supports accelerators developed in RTL, generated with HLS from C, C++, or SystemC, and generated from machine-learning models with HLS4ML. The accelerator socket provides services for address translation, DMA, configuration registers, and coherence. Pre-designed third-party accelerators can also be integrated when they use a standard interface such as the Advanced eXtensible Interface (AXI) [13, 11, 24].

The *network-on-chip* connects all tiles through six independent physical planes. The use of multiple planes separates different types of traffic and allows each plane to use simpler routers. In ESP, three planes carry coherence messages, two carry DMA requests and responses, and one carries memory-mapped register accesses and interrupts. The NoC is synchronous and uses dimension-order look-ahead routing, with a latency of one clock cycle between adjacent routers. Each tile socket contains proxies that convert bus requests into NoC packets and NoC packets back into bus transactions. These proxies allow components in different tiles to communicate as if they were connected to the same bus [13, 25].

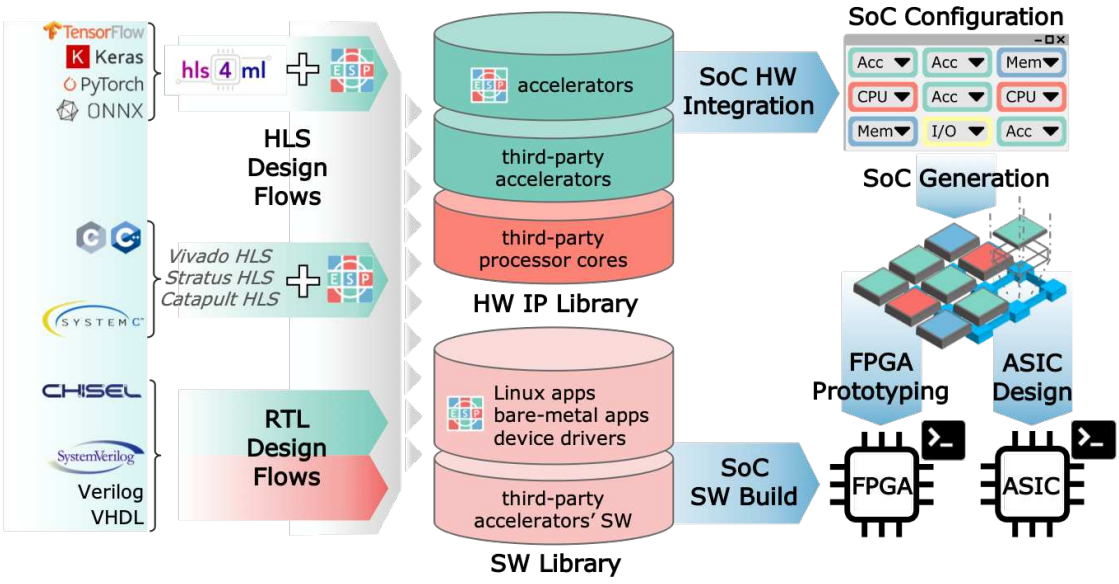


Figure 2.6: ESP design platform flows overview. Reproduced from Mantovani et al. [11].

The ESP methodology is organized around two main flows, as summarized in Figure 2.6. The *accelerator flow*, shown on the left side of the figure, adds reusable hardware accelerators to the ESP IP library through three design paths. The traditional RTL flow uses Verilog, VHDL, or Chisel to describe a cycle-accurate hardware design. The commercial HLS flow uses ESP in-house automation scripts together with Xilinx Vivado HLS, Mentor Catapult HLS, or Cadence Stratus HLS to generate accelerators from C-like languages, like C++ or SystemC. The open-source *hls4ml* flow generates synthesizable accelerators from machine-learning models developed with Keras, TensorFlow, or PyTorch [26]. These higher-level flows reduce the amount of low-level RTL that must be rewritten when exploring and optimizing alternative accelerator architectures. This also reduces the risk of introducing bugs during large design changes [11, 13].

The *SoC flow*, shown on the right side of the figure, realizes the complete SoC after its hardware and software components have been designed. It selects and combines processors, accelerators, memories, and peripherals, and generates the files needed to integrate the hardware with drivers and applications. This enables hardware–software co-design across the complete system. The resulting design can be simulated and implemented on an field programmable gate array (FPGA) for rapid prototyping or onto the ASIC flow for silicon fabrication [11, 24, 13]

2.3 ESP ASIC Flow

Historically, ESP has focused on system-level design, up to RTL level, with SoC prototyping targeting FPGAs. Thanks to the contributions described in Cassel Dos Santos's ICCAD 2022 paper [24], ESP is now able to support ASIC implementations. This was possible by many enhancements in the ESP architecture and it was supported by the collaboration of multiple universities and companies leading to two SoC tapeout, namely EPOCHS-0 and EPOCHS-1 [27, 28].

Hierarchy restructuring changed how the tiles and the NoC are partitioned for physical design. At first the NoC and the tiles existed as separate entities. The first restructuring placed one router for each NoC plane inside every tile wrapper. However, a loss of NoC timing performance appeared in both EPOCHS-0 and EPOCHS-1 chips. Although the NoC met timing at 800 MHz at tile level, the complete SoCs closed timing at 500 MHz. These results showed that the delay between two adjacent routers heavily depended on how both tiles were physically implemented. Additionally, an incorrect NoC constraint or power-domain definition in one tile could affect the communication network of the complete SoC. In the final hierarchy change, the routers are removed

from the tiles. Each tile contains its processing element and socket, while the NoC router is implemented as a standalone macro and repeated at the top level. This gives every router instance the same physical implementation, reduces design variation, and leaves the top level mainly responsible for connecting the tile and router macros and distributing the global clock [13, Sec. 5.2].

Control and status registers (CSRs) provide additional runtime configurations required by ASIC implementations. Each tile contains a register block connected through an Advanced Peripheral Bus (APB) interface and accessible from ESPLink or software running on the SoC. These registers configure components such as the digitally controlled oscillator (DCO) and the I/O pads. They also store the tile identifier, which is assigned over the NoC after reset instead of being fixed as an RTL parameter. This allows the same physical tile implementation to be reused throughout the physical design. Additional counters record events such as NoC traffic, coherence messages, cache accesses, and external memory accesses [13].

Local clock generation supports independent clock and voltage control for the tiles and the NoC. A DCO can be instantiated in each tile, while the DCO for the NoC is placed in the IO tile. Each DCO is configured through a dedicated register, and the local reset remains asserted until the oscillator has run for the required number of cycles. This organization enables a communication-synchronous, globally asynchronous, locally synchronous (CS-GALS) system: the tiles operate at independent frequencies and communicate with the synchronous NoC through asynchronous interfaces. The later flow also implements a technology-driven clock tree for the standalone routers, using characterized buffer spacing and routing layers to control the clock paths between neighboring routers [13, Secs. 3.1 and 5.2].

Memory integration maps the memories used by ESP to the macros available in the target technology. In constrast with FPGA prototyping, where BRAMs with different sizes are configured with the memories required by the ESP architecture, physical static random-access memory (SRAM) macros must be instead used. This includes caches, shared local memory (SLM) and private local memory (PLM), the boot ROM, and Ethernet register files. A memory list describes each memory type, the selected macro, and its interface parameters. The integration flow uses this list to generate ESP-compatible wrappers and the mapping file required by the design hierarchy. It also generates standalone testbenches so that each wrapper can be checked before full-system simulation [13].

Since ESP-compliant and open-source double data rate (DDR) controllers are challenging to find, it was decided to implement the *FPGA link for DRAM access*, providing external

memory access. A bridge in each memory tile connects the SoC to an FPGA board that contains the controller and the external dynamic random-access memory (DRAM). The bridge uses a credit-based protocol and transfers data as ESP NoC packets, allowing the FPGA host to reuse components from the chip design [13].

SoCProbe for single-tile test provides an access path that does not depend on the normal operation of the NoC. The unit is integrated within the tile socket and uses a four-pin serial interface to send and receive NoC flits. In tile-test mode, it disconnects the tile from the router and communicates directly with the tile while preserving the latency-insensitive interface. The latest SoCProbe architecture also defines a NoC-test mode, which disconnects the tile and connects SoCProbe to the router. The two modes allow tiles and routers to be tested separately without changing their internal logic [13, Secs. 3.1 and 5.1].

2.3.1 Tile-based hierarchical flow

As mentioned in Section 2.1, the digital ASIC design follows a well-defined flow, similar across multiple EDA vendors. In constrast to this, the ESP ASIC flow proposes a different, hierarchical methodology that provides productivity gains through flexibility, robustness and scalability. The overview of the flow is shown in Figure 2.7.

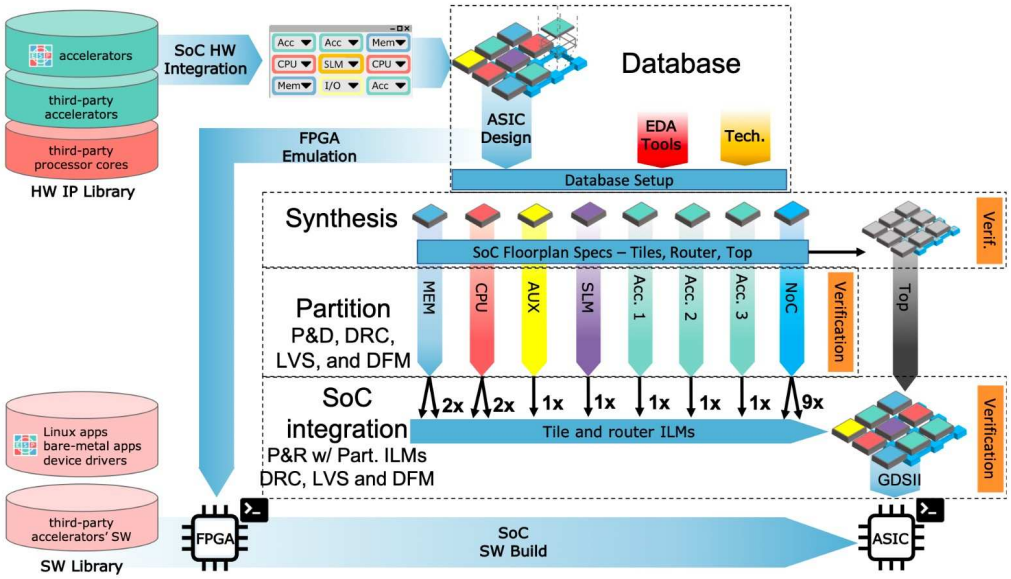


Figure 2.7: Methodology proposed for agile ASIC development. Reproduced from Cassel Dos Santos [13].

As a preliminary setup step, the SoC configuration is selected and the database is populated with technology files and library data. Given that ESP natively supports

FPGAs, this capability can be leveraged in the ASIC flow as an FPGA emulation of the target SoC configuration, enabling early-stage design space exploration and preliminary testing of the software stack.

The core of the flow is the use of *design partitioning*, i.e. dividing the top-level design into smaller components, which for ESP are the tiles and each NoC router interface. This presents several advantages to top-level flattening: it parallelizes the implementation, allowing the workload to be split across different less powerful machines, shortens execution time of the flow, and it eases detection and fix of issues and bugs. However, the main drawbacks are challenging timing closure and limited top-level optimization. For the first, the NoC router interface is treated as a unique block and shared among all the tiles, the interface delays from tile-to-tile can be similar with an adequate choice of timing and physical constraints, therefore facilitating timing closure. For the latter, an appropriate floorplan can reduce significantly the penalty for treating each tile and NoC router as a standalone component.

Because of partitioning, each tile type, the NoC router, and the toplevel present different *timing constraints*. Toplevel constraint file shares external delays, with a minor discount for I/O pads delays, and uses the same clock relationship used across all its components. In particular, the NoC clock frequency and the tile's IP clock frequency must be set for each subcomponent. Based on the clock strategy their relationship can change. In addition to that, the enabled peripherals and the number of memory tiles impact the constraints, because each of these interfaces has its own clock that interacts with the external environment.

Then, each component first undergoes logic synthesis for an early estimation of timing, power, and area. After that, physical constraints can be estimated using post-synthesis data, allowing each component to move to a floorplanning stage in the physical design tool. The results from the logic synthesis and floorplanning allow the synthesis of the top-level. Moreover, the gate-level netlist and the delay information file (usually in an SDF format) generated for each block and top-level enable post-synthesis verification steps, namely equivalence checking and gate-level netlist simulation.

Alternatively, one can continue the flow with the physical design of each component, leaving top-level synthesis for a later step. The physical design stage produces the layout, block-level views (usually in ILM format), the post-PnR netlist, the delay information file, and the parasitic data. Among these, the physical and block-level views of each component permit top-level physical design and the integration of each component within it. De facto, this restricts the top-level physical implementation solely to the logic surrounding each tile and NoC router.

The final layout of each component and the top-level are physically verified both at the abstract view (LEF-based) and at the full-layout level (GDSII-based). For the latter, physical verification collaterals (such as those for DRC and LVS) are located in a separate section of the flow and are therefore not directly integrated into the main pipeline. Future releases will integrate these steps into the main flow.

2.3.2 Project structure

A unified database infrastructure has been implemented to support multi vendor EDA tools, multi-technologies, and multiple SoC designs of different configurations and size. The directory structure, shown in Figure 2.8, highlights the four subdirectories from the project root: *ESP* repository, *technology*, *ESP EDA*, and *design*. This layout was chosen since keeping it outside the ESP main repository minimizes the risk of accidentally pushing sensitive data.

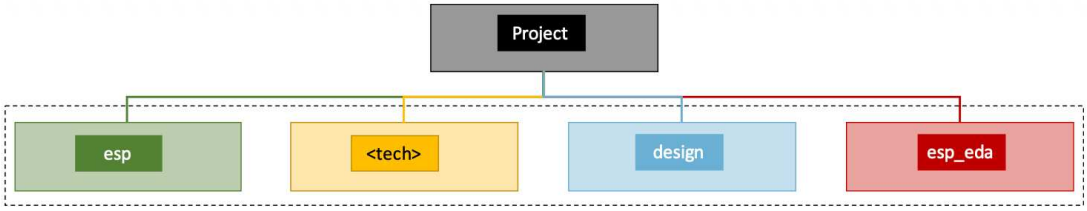


Figure 2.8: Database directory structure for the ESP ASIC methodology.

The ESP directory provides the RTL files used for logic synthesis and configuration files for the generation of the *technology* and *design* folders, while the *eda* folder is kept as a private repository. This serves as the backbone for the pre-RTL stage by providing all the necessary HDL files across multiple sources, either HLS-generated or RTL-engineered, and also as a bridge for the ESP ASIC flow.

The *technology* directory contains all technology files, libraries data and additional metadata for loading the database. This includes RTL wrappers used for memories, pads and DCO integration, timing, physical and electrical views of standard cell libraries, power management kits and IPs, and technology files used for knowing the metal stack of the technology, RC extraction data and corners, on-chip variation models. It is suggested to use soft-links rather than directly copying sensitive data from the PDK and library vendors.

At the time of this work, the ESP ASIC flow natively supports a FinFET-based 12 nm technology using commercial third-party libraries. The names of the PDK and library

vendor are covered by signed non-disclosure agreements and cannot be reported here. The original library and corner names are omitted for the same reason. Descriptive aliases are used when these names are needed to explain the flow or report results.

The *ESP EDA* directory consists of TCL scripts consumed by the vendor EDA tools, *Make* directives, and Python-based automation scripts. The latter is used both for the flow's working directory setup and for the floorplanning auto-generation used for Physical Design, based on the SoC configuration.

From ESP's perspective, the *design* folder is mimicking one of the *socs* subdirectories, used for FPGA prototyping. However, it primarily differs by the use of the *ESP SoC* subdirectory, generated through the Python scripts inside *ESP EDA*. The structure of this folder is highly hierarchical, containing the following subdirectories: *config*, consisting of system-level definitions, including SoC constraints, floorplan information and flow configurations, and design constraints information consumed by the flow, i.e. clocks and corners definitions; one subdirectory for each unique tile type, meaning if there is a replica of the same tile it is not duplicated but only referenced to the first named tile sharing the same type; one subdirectory for the NoC router and lastly the toplevel subdirectory. Then, in the lower hierarchical level, EDA tools' reports, outputs and logs are stored for each component, i.e. the tiles, NoC router and toplevel, based on the flow stage. It includes also generated timing constraint, power intent and I/O locations files.

2.3.3 Features of the methodology

Clock strategies. The ESP ASIC flow supports three clock strategies: *dual external*, *single DCO* and *multi DCO*. The *dual external* mode uses only one external clock and propagates it throughout the entire SoC; in *single DCO* mode, a single DCO, located in the IO tile, is used for clocking the NoC and the tiles; the *multi DCO* approach fully realizes the ESP vision by instantiating a DCO in every tile, along with a separate DCO for the NoC, which is also located in the IO tile. Each DCO includes a fallback option for external clocking. For each tile, this external clock comes from the NoC clock, while for the NoC DCO the fallback options lies in the actual external clock that is provided as a standalone top-level I/O.

Power strategies. The latest update of the methodology partially enables multi-supply voltage (MSV), while leaving room for future power shut-off (PSO) implementation. The default option is a single *global* power domain applied to each tile and the NoC. However, the use of a CS-GALS design defines natural boundaries for the different

power domains. In this context, the power domains follow the *multi DCO* clock strategy: each tile and separately the NoC can operate at different adjustable power domains, via a configurable power constraint file. Although EPOCHS-1 has been proven to have multi-power domain capabilities, there is currently ongoing work to enable this feature in the flow following the latest major methodology refactoring.

Automatic design constraint generation. The current flow supports automatic generation of design constraints, in SDC format, based on the flow and SoC configuration. Currently, the clock strategy selection has the most impact on clock relations, along with additional constraints related to enabled peripherals. The selection of these parameters at the SoC configuration level yields different combinations of design-dependent clock settings. The external interfaces are set according to protocol standards, except for the FPGA memory link, which varies depending on the test environment. All other technology-dependent constraints, such as driving cells, clock transitions, and maximum fanout, are set once and applied across all components and the top-level. The settings are written in a way that allows them to be reused across different designs targeting various technologies.

SoC automatic floorplanning. The methodology automatically derives the dimensions of the routers, tiles, core, and die from post-synthesis area estimates, the SoC configuration, and technology-specific placement and routing grids. It also assigns aligned pins, places the tiles and routers, checks whether the design is limited by the available pad or bump capacity, and generates floorplan-aware placements for the GPIO cells and bumps. For designs with multiple power domains, supply bumps are distributed according to the estimated power and physical location of each domain, while the generated RDL connections and core rings support power delivery. The resulting floorplan metadata is used by the physical implementation scripts, and CSV files allow the bump assignments to be adjusted for packaging or board constraints before regenerating the floorplan [13, Sec. 6.2].

Chapter 3

Robustness and Configurability of the ESP ASIC Flow

The ESP ASIC flow already supports different system and implementation choices. However, these choices pass through several configuration files, generators, and tool scripts. A new release or a different configuration can therefore expose assumptions that were valid only for the setup used when part of the flow was written. This chapter addresses these problems and prepares the flow for the standalone reuse discussed in Chapter 4.

The chapter first defines how the flow is validated and introduces the main SoC and ASIC configuration options. It then describes the problems that limited their propagation through the flow. The changes follow the same order in which these problems were encountered: the initial ASIC preparation was corrected, the Python scripts were refactored, the timing constraints were made conditional, and the selection of PVT corners was extended.

The final section brings these changes together through a set of regression tests. The results are limited to the tested release, configurations, tools, and technology, following the validation levels defined at the beginning of the chapter.

3.1 Multi-Level Validation of the ESP Design Flow Across Releases

The ESP design flow changes across releases, since a release may modify the RTL, configuration files, scripts, dependencies, or tool requirements. The platform and the implementation flow are also maintained in separate repositories, so a change in one repository can break an assumption made in the other. For this reason, a flow that works with one release may not work with the next one.

Validation is therefore linked to a specific ESP release, SoC configuration, tool environment, and technology setup, and a previous result cannot be reused as proof for a new release. Instead, the flow must be executed again from its first required stages, using representative configurations to check its main paths.

When a stage fails, its cause is identified and corrected, and the affected stages are then executed again. This process detects regressions and outdated assumptions that may otherwise remain hidden. Since a broken stage prevents the flow from producing its intended output, its functional correctness cannot be considered established. Correcting the problem restores this function for the tested case and improves the robustness of the flow, although the result still does not cover releases or configurations that were not tested.

3.1.1 Validation Depth

Not every change requires validation up to fabrication signoff, because the required depth depends on the change and on the intended use of the design. For example, a software change may only require synthesis, while a change that affects, for example, the technology files should also be checked during physical design. A design intended for fabrication instead requires the complete signoff procedure. For this reason, this thesis uses the three validation levels shown in table 3.1. The levels are cumulative, so each one includes the previous level and adds later stages of the flow.

Level 1 checks configuration handling, RTL processing, and logic synthesis, and it is suitable for many changes to Python code, Make automation, source discovery, and synthesis scripts. Its main output is a valid gate-level netlist. However, successful synthesis alone does not prove all hardware behavior, so simulation and other consistency checks are still required when they are part of the tested flow.

Level 2 continues through floorplanning, placement, CTS, routing, and post-route analysis, and it is needed when a change can affect timing or physical implementation. This level can support architectural studies and the analysis of power, performance, and area, but it does not show that the layout is ready for fabrication.

Table 3.1: Cumulative validation levels for the ESP design flow.

Level and scope	Completion evidence	Claim supported within the evaluated scope
1. Logical-flow validation: configuration and RTL processing through logic synthesis	Elaboration and synthesis complete, the main checks pass, and a gate-level netlist is generated	The front-end and synthesis flow works for the tested release, configuration, constraints, libraries, and tools. Preliminary metrics can be evaluated
2. Physical-implementation validation: Level 1 through floorplanning, placement, CTS, routing, and post-route closure	A routed design is produced and meets the stated timing and implementation criteria	The flow can produce a physical implementation for power, performance, and area evaluation under the stated conditions
3. Tapeout-readiness validation: Level 2 plus the signoff checks required by the target process and fabrication methodology	Extracted timing, DRC, LVS, EMIR, power, thermal, and manufacturability checks meet the signoff criteria	The tested design is tapeout-ready for the stated technology and signoff conditions

Level 3 adds the signoff checks required by the target process, including physical verification, reliability analysis, and manufacturability checks. The exact checks depend on the PDK, the foundry, and the signoff methodology, so this level is needed for technology changes and for designs intended for fabrication.

3.1.2 Release-Dependent Failures Observed in Practice

The first execution of the flow in this work exposed several failures. Some automation scripts and Make directives contained typographical errors, while some Python files contained invalid syntax and crashed during execution. Other scripts used legacy

vendor-tool commands that had been deprecated. Because of these problems, the flow stopped before the design results could be evaluated, so the errors had to be identified and corrected first.

At the same time, the workstation environment did not support the updated ESP release. For example, the release required ModelSim 2021 or newer, but the provided account only included ModelSim 2019. As a result, the required simulation stages were blocked by the version mismatch. A newer operating system was installed to create a compatible environment, after which the main computer-aided design and EDA tools were reinstalled. Their environment settings and licenses also had to be configured again.

These problems show why validation must be repeated for each new release. Even when the flow worked before, it may fail because of a script change, a dependency, or a different tool version. Re-executing the flow reveals the broken stage and makes it possible to correct the problem. Once corrected, the flow is validated again for the tested release and configuration. Repeating this process across selected releases then provides evidence of robustness across those releases.

The final claim is limited by the completed validation level, as well as by the tested configurations, tools, and technology. Level 1 supports a claim about the logical flow, while Level 2 supports a claim about physical implementation. Only Level 3 supports a claim of tapeout readiness. The regression matrix in Section 3.8 applies these limits to the validation cases used in this thesis.

3.2 ESP SoC Configurations

ESP aims to provide flexibility, scalability, and robustness. At system level, the ESP graphical interface offers several configuration options. Users can then build a SoC based on the target application. The available choices also change between FPGA prototyping and ASIC implementation.

The options described here refer to release 2026.1.0. A small note: when a range is given, it represents the discrete values given by the graphical interface. In most cases, these values increase by powers of two.

Figure 3.1 shows where the main configuration choices enter the flow. Each grey area represents one configuration file. The red boxes are individual settings, while the white boxes are groups that can be expanded into further settings. Only the groups and individual choices most relevant to this work are shown. The first file belongs to the

ESP SoC Generator. After it is configured, `make esp-xconfig` generates the ESP top level. From there, `make asic-setup` creates the *ESP SoC (ASIC)* working folder, which contains the second file for SoC Flow Control.

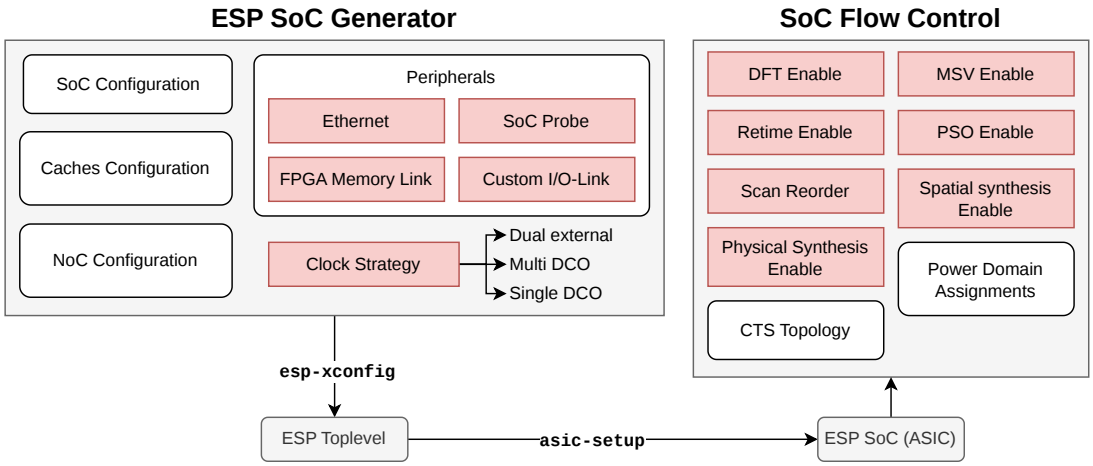


Figure 3.1: Main configuration files and setup steps of the ESP ASIC flow. Red boxes show individual settings, while white boxes show expandable groups of settings.

Within the ESP SoC Generator, the first group concerns the general SoC configuration. The processor can be Ibex, Leon3, or the older Ariane core. Work to integrate the newer CVA6 architecture is ongoing. The shared local memory can also be selected for each tile. Its default size is 512 KiB, with values from 64 to 4096 KiB.

The peripheral configuration includes UART, Ethernet, SoCProbe, the I/O link, and the FPGA memory link. UART cannot be disabled in the current flow, while the other interfaces are configurable. The FPGA memory link uses 64 bits by default and supports values from 32 to 512 bits. The I/O link uses 16 bits by default and supports values from 8 to 32 bits.

The clock strategy can be dual external, single DCO, or multi DCO. These strategies were introduced in Section 2.3.

The cache hierarchy can be enabled or disabled. Its implementation can be from the ESP RTL, SPANDEX HLS, or ESP HLS, with ESP RTL being the default. The processor and accelerator caches use 512 sets and 4 ways by default. The last-level cache uses 1024 sets and 16 ways. The number of sets ranges from 32 to 8192. The processor and accelerator caches support from 2 to 8 ways, while the last-level cache supports from 4 to 16 ways.

The NoC configuration first defines the number of rows and columns in the tile grid. The widths of coherence planes 1, 2, and 3 and DMA planes 4 and 6 can also be

selected. Their default width is 64 bits, with values from 32 to 1024 bits. The MMIO and interrupt plane 5 instead has a fixed width of 32 bits. If a selected width is larger than the processor word size, the cache hierarchy must be enabled. The processor word size is 32 bits for Ibex and Leon3 and 64 bits for Ariane. Finally, multicast can be enabled on the DMA planes. The maximum number of multicast destinations and the router FIFO depth are also configurable.

These choices describe the structure of the generated SoC. The SoC Flow Control file then adds the implementation-specific settings. In release 2026.1.0, they are selected in a TCL-based configuration file rather than in the graphical interface. Moving them into the interface remains future work.

The TCL configuration sets the clock periods, the power domains, and other implementation parameters, including settings for CTS. More options can be set for each tile type, the NoC router, and the top level. The main options are described below.

The same file contains switches for DFT, retiming, physical-aware synthesis, spatial-aware synthesis, MSV, PSO, and scan reordering. These options do not all have the same level of support.

DFT enables scan-chain insertion at block level or at the top level. Its support is still partial because the top-level connection strategy is missing. Scan reordering depends on these chains and is also disabled for now.

Retiming allows registers and flip-flops to be moved during synthesis. It can improve performance, power, or area, but it is disabled by default. It must be used with care because it can make debugging and equivalence checking more difficult. Incorrect constraints can also make the resulting timing invalid.

Physical-aware and spatial-aware synthesis both use physical information during logic synthesis. In the current flow, they require a first logical-synthesis pass. This pass estimates the area of each component before its floorplan is generated. Spatial-aware synthesis is mainly intended for the Cadence-based flow through its iSpatial technology.

MSV support must be enabled together with the selected power domains. It is only partially integrated because the flow does not yet generate the UPF description automatically. PSO, which would switch off parts of the SoC when they are not used, has not yet been implemented.

Among the system-level choices, the clock strategy has the largest effect on the ASIC flow. It changes the clocks in the SoC and their synchronous or asynchronous relations

at the CDC boundaries. Peripheral selection also matters because signals can appear or disappear from the IO tile.

The full configuration space was not tested exhaustively in this thesis. Instead, the validation focuses on selected configurations and parameters, as reported in Section 3.8. Several TCL-based ASIC-specific settings are also only partially integrated and still require further implementation before they can be validated.

3.3 Flow Scalability Barriers

As ESP evolves, a new feature can change the RTL hierarchy, the ports of a tile, the clocks, or the required implementation inputs. The flow must follow these changes across its Python generators, TCL scripts, constraints, and physical design files. In this context, a configuration choice should propagate automatically through the flow. For example, disabling a peripheral should not require manually removing its signals from multiple Python and TCL scripts.

The first problems appeared before running the private ESP ASIC flow. The public technology-integration procedure prepares the project, integrates the technology-dependent components, and verifies the generated SoC through RTL simulation. The private implementation flow starts only after these steps [13, Appendix A]. Therefore, the ASIC integration refactoring discussed in this chapter does not concern the introduction of a new technology. It concerns the preparation needed before the implementation flow can be used.

Following this procedure exposed typographical errors, scripts that were difficult to read, and crashes during execution. The RTL simulation could also hang when using the multi-DCO strategy. These problems had to be handled before starting the work on the private ASIC flow. The first refactoring step therefore focused on making this preparation more reliable.

Once the design reached the implementation flow, the structure of the Python scripts became another obstacle. The code was difficult to read, maintain, and extend, in addition to some critical bugs found, which limited the scalability of the flow. The scripts were therefore refactored before revising the generation of timing constraints. The specific changes are described in Section 3.5.

This work then made it possible to refactor the generation of SDC. The original constraints had been developed mainly around the multi-DCO strategy. The dual external and single DCO strategies exposed timing violations and incomplete assumptions about

the clock relations. The constraint generation therefore had to follow the selected clock strategy and the actual ports of the design.

After the design and constraint inputs for synthesis were ready, the next extension concerned the selection of different PVT corners.

3.4 ASIC Integration Refactoring

The ESP ASIC methodology requires a preparation phase before the private implementation flow can be used. This phase creates the project, integrates the technology-dependent components, generates the SoC, and checks it through RTL simulation [13, Appendix A]. This section concerns corrections to that procedure, not the porting of the private flow to a new technology.

3.4.1 Project and Component Preparation

The first step clones the ESP repository, names the design and technology folders, and runs the directory generator. This creates the ESP, technology, and design directories introduced in Section 2.3.

Memory integration maps the available macros to the interfaces expected by ESP. A memory-list file gives the type, name, depth, word width, area, and port type of each macro. A Make target generates the wrapper skeletons, which are then completed with the required signal mapping.

The procedure also generates a testbench for each wrapper. These tests initially failed because their clock period and access times did not satisfy the timing checks in the vendor models. The testbenches were adjusted manually with compliant values. The generator was left unchanged because these values depend on the selected technology, so their automatic selection remains open.

Pad integration follows a similar procedure. A pad-list file defines the direction, macro name, and location, while a second list assigns the SoC signals to horizontal or vertical pads. A typographical error in the corresponding Make target prevented this step from running and was corrected. The default generator still assumes one memory tile and all peripherals enabled.

The DCO integration uses technology-independent structural models with generic cell names. These names must be mapped to the available standard-cell library. The guide

describes the DCO as a hard macro with timing, physical, delay, and layout views. However, the current methodology keeps it as a soft macro and preserves it during logic synthesis. Generating a complete hard macro was outside the scope of this thesis.

3.4.2 SoC Generation

After integrating the memories and pads, a common SoC configuration was selected for the following work. It uses a 2×2 grid with one IO tile, one Ariane processor tile, one FFT accelerator tile, and one memory tile. The processor has an L2 cache, while the memory tile has an LLC. The FFT accelerator uses 32-bit data and a 64-bit DMA interface without an optional L2 cache. The configurable NoC planes use a width of 64 bits.

This configuration does not contain all five ESP tile types. Therefore, a temporary 3×2 configuration added an empty tile and a shared-local-memory tile for the dedicated shared-memory tests. The initial clock strategy was dual external. The other strategies were considered later to check that the preparation procedure was not limited to this initial choice.

The FFT accelerator was synthesized with Cadence Stratus, which is the supported HLS tool for this ASIC preparation procedure. Stratus crashed when the standard-cell libraries used advanced CCS or ECSM timing models. For the HLS pass, the symbolic links in the technology folder were temporarily changed to point to NLDM timing views instead. The original links were restored after the accelerator had been generated. This was a tool workaround rather than a change to the library data.

Release 2026.1.0 also introduced a new graphical interface based on Python and Tkinter. Its ASIC configuration adds a choice for the clock strategy. However, the corresponding widget expected a string while the strategies were represented by numeric indexes. This type mismatch caused the interface to crash. The fix separated the label displayed by the menu from the numeric value stored in the SoC configuration, added an explicit mapping between them, and updated the dependent configuration when the selection changed.

3.4.3 RTL Simulation

The main-system simulation checks the processor, IO, and memory tiles. It loads the bootloader through the IO tile, releases the processor, and exercises accesses to the

memory tile and external memory. A successful execution prints the message `Hello from ESP!` and terminates.

This step first required the workstation update described in Section 3.1, because the installed ModelSim version was not compatible with release 2026.1.0. After the environment was restored, the dual external and single DCO simulations completed. The multi-DCO simulation instead remained active without completing the processor reset sequence.

The hang was caused by the interaction between the structural DCO model and the testbench timing. SDF annotation was disabled, and the default DCO configuration produced a slow IO clock. Its lock sequence required 1024 cycles, but the testbench sent the boot commands before the IO link had left reset. The commands were discarded, so the processor reset sequence could not finish.

Two behaviors were added for this case. The simulation now reports an error when the structural DCO is selected without an SDF file. A behavioral DCO model can instead be used for functional RTL simulation. This second path checks the tile functionality without requiring the complete structural timing model, which is more suitable during early preparation.

The remaining simulations covered the bare-metal accelerator test and the shared-local-memory read and write test. The shared-memory test completed without further corrections. The accelerator test was compatible with the FFT variant using 32-bit data and a 64-bit DMA interface, so this variant was selected. The test itself was not modified because extending it to the other generated accelerator variants was outside the maintenance work on the ASIC flow.

After these corrections, the project could complete the public preparation procedure and provide the inputs required by the private implementation flow. The next step concerned the structure and maintainability of the Python automation used within that flow.

3.5 Python Refactoring

As described in Section 2.2, ESP combines an architecture with a design methodology. The Python scripts are part of this methodology because they translate the selected SoC hierarchy and configuration into files used by the implementation flow. Their structure therefore affects how easily the flow can follow changes in the platform. Although the

hardware architecture is modular and scalable, the original Python automation was difficult to maintain and extend.

Several modules mixed function definitions with code that ran as soon as the file was loaded. File handling relied on repeated `open()` and `close()` calls, while paths and generated text were often assembled through long string concatenations. The same TCL-writing patterns appeared in several generators, together with large conditional blocks, unused imports, commented experiments, and hardcoded examples. Taken together, these choices made the scripts difficult to read and made later changes harder to apply consistently.

The first changes gave the modules a clearer execution structure. Executable code was moved into `main()` functions and protected with `if __name__ == "__main__"`, so importing a module no longer started file generation or other flow actions [29]. File operations were moved to `with open(...)` blocks, which close their files when the block ends, including when an error occurs [30]. Path construction was also moved to `pathlib.Path`, and executable scripts received a portable Python shebang. Unused imports, dead branches, and old examples were removed at the same time.

The main structural change was the introduction of the `TclWriter` class. Before this change, each generator built Tcl files through its own sequence of `write()` calls. `TclWriter` now provides common methods for adding lines and multiline blocks, writing headers and sections, working with database variables, and producing the final Tcl text. This class-based approach makes the generators more modular because common behavior is kept in one place. It is also easier to extend: a useful method can be added once and then reused by every generator. The same structure is used later for the conditional constraints discussed in Section 3.6.

Figure 3.2 shows a small example of this structure. The `sdc_clock()` method builds a `create_clock` command, while `db()` inserts the reference to the clock period stored in the design database. The resulting command is enclosed in a line through a `line()` method, and `render()` produces the final `tile.sdc` file. The generator can thus describe the clock through these methods instead of rebuilding the complete command as a string each time.

Other changes improved readability and consistency across the scripts. F-strings replaced many concatenations and `format()` calls in paths, messages, and generated text [31]. Black was configured and applied to keep indentation, line wrapping, spacing, and collection formatting consistent. This follows the general Python guidance on readable and consistent code [32]. Comments were kept where they explained the purpose of the code, while stale comments and repeated formatting logic were removed. One correction

was not only a cleanup: a call written as `line.split` instead of `line.split()` could stop configuration parsing and was fixed.

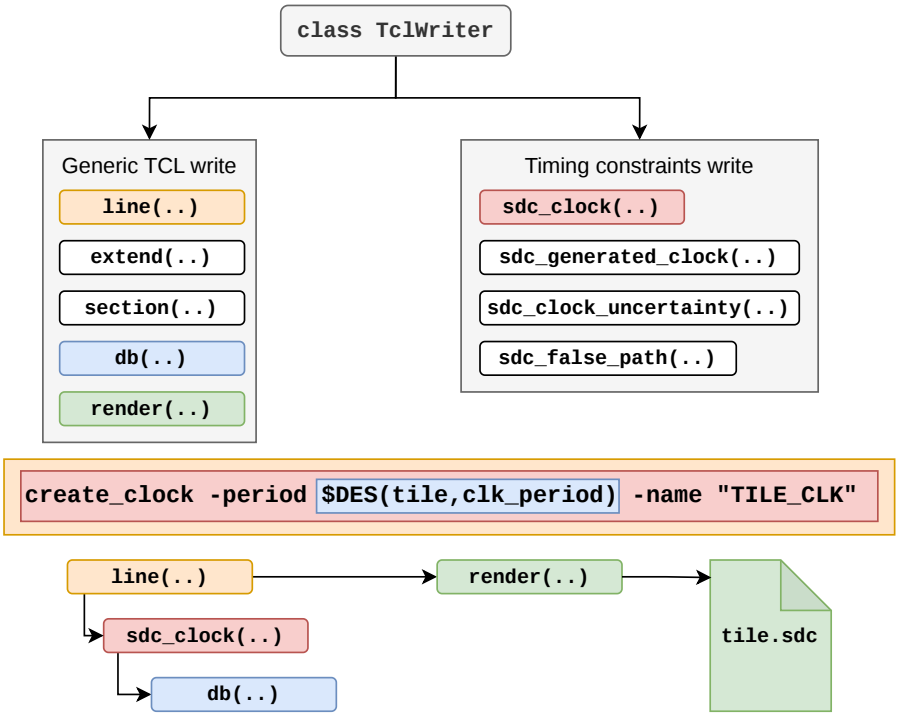


Figure 3.2: Generation of a `create_clock` constraint with the `TclWriter` class. Matching colors connect each part of the command to the method that produces it.

Overall, the refactoring made the modules safer to import and easier to read, while giving the Tcl generators a common interface that can grow with the flow. This is important for ESP: a scalable and modular architecture also needs automation that can be updated without repeating the same change across many scripts. The changes were checked by running the affected scripts and comparing their generated files with the previous outputs. The refactored scripts produced the same results. This check is narrower than the validation levels defined in Section 3.1: it shows that the Python refactoring did not change the generated files, but it does not make a new claim about synthesis or physical implementation.

3.6 Conditional SDC Constraint Generation

The Python refactoring provided a cleaner base for revising the generated timing constraints. This second change was not limited to replacing repeated `write()` calls. The original generators contained large blocks of repeated SDC text, while several

constraints still followed assumptions made for the multi-DCO strategy. As a result, changing the clock strategy or disabling an interface did not always produce constraints that matched the design.

The work covered the common clock definitions, the constraints for each tile, and those for the standalone NoC router. The `TclWriter` class was extended with methods for clocks, generated clocks, uncertainty, cycle percentages, and false paths. Repeated divider chains inside the DCOs were also generated through common functions instead of separate commands. This kept the shared parts compact, while leaving the generators free to handle the differences between tiles and clock strategies. The result was a major rewrite of the generators, but their common parts became shorter and easier to follow.

The new generators use the SoC configuration directly. The selected clock strategy decides which periods and clock properties are independent and which ones are inherited from another clock. The JTAG, Ethernet, and I/O-link enables control their related definitions. The same information is passed to the tile and NoC generators, where it changes the clocks, case analysis, interface delays, and false paths. The main differences are summarized in Table 3.2.

Table 3.2: Effect of the clock strategy on the generated tile and NoC constraints.

Clock strategy	NoC and tile clocks	Generated constraints
Dual external	Supplied through the external clock path and treated as synchronous	The DCO clock chains are omitted, and no false path is added between the NoC and tile domains
Multi DCO	Supplied by separate DCOs and treated as asynchronous	Both divider chains are generated, together with false paths between the NoC and tile domains
Single DCO	Supplied by the I/O tile DCO and treated as synchronous	The NoC divider chain is generated, while the per-tile chains and the NoC-to-tile false paths are omitted

The three cases now follow the actual clock structure. In multi-DCO mode, the generated file keeps separate period and clock-property entries for every tile. In the other two modes, the tile entries follow the NoC clock. Only multi-DCO mode creates an asynchronous boundary between the tile and NoC clocks. The active clock domains are collected into groups, and bidirectional false paths are generated between each pair of different groups. This replaces the long list of individual cases in the previous generator and includes optional interface clocks only when they are enabled.

Port handling was revised at the same time. Resets, clocks, clock-status signals, local NoC ports, and external interfaces are placed in explicit collections before applying the

generic input and output delays. The standalone router also distinguishes its north, south, east, and west links from the local tile interface. Therefore, the directional links are timed with the NoC clock, while the local interface follows the tile clock. Dedicated constraints are added for JTAG, the FPGA memory link, Ethernet, and the I/O link. UART remains asynchronous and is excluded through false paths.

A relevant missing constraint concerned the Ethernet MDC output. The old generator did not create a clock for this signal, even though the Ethernet block derives it from the I/O tile clock. The new generator first obtains the division factor from `mdcscaler`:

$$\texttt{mdcdivider} = 2(\texttt{mdcscaler} + 1), \qquad T_{\mathrm{MDC}} = T_{\mathrm{TILE}} \times \texttt{mdcdivider}.$$

For the tested configuration, `mdcscaler` is 490. The division factor is therefore 982, and a tile period of 1.7 ns gives an MDC period of 1.6694 μs. The generator creates `MDC_CLK` at the MDC output and keeps its relation with the source tile clock. Uncertainty, transition, and false-path generation were extended to include this clock as well. Representing it as a generated clock lets timing analysis preserve its relation with the source clock [18].

The revised port collections also prevented clock-monitor and status signals from receiving generic datapath delays. Other timing violations exposed problems in the RTL rather than missing exceptions. In particular, a crossing between the NoC router and the local tile required synchronizers, while an incomplete assignment in the memory tile inferred a latch. These paths were corrected in the RTL; they were not hidden with additional false paths.

The generated constraints were tested through Level 1 using selected cases that covered all three clock strategies, the tile types, the standalone NoC, and different peripheral combinations. These runs completed without timing violations. The complete validation matrix is reported in Section 3.8. Conditional generation of the top-level SDC was not completed and remains future work. With the tile and NoC synthesis inputs in place, the next step was to extend the selection of PVT corners.

3.7 Configurable PVT Corners Selection

The flow already used the MMMC structure introduced in Figure 2.3. For each run, it generated the corner definitions and the setup and hold analysis views. The problem was how the Python script found the available corners. It also offered no simple way to run the flow with only a selected part of them.

Originally, the script opened every Liberty file in the technology library folder. It read the operating condition name and its process, voltage, and temperature values, and then grouped entries with the same values. The first name found for each combination was kept as the corner name. This worked while the folder contained libraries that used the same naming rule.

The problem appeared when the standard-cell libraries were loaded together with the SRAM and GPIO libraries. These libraries can describe the same main operating condition with different names. In particular, SRAMs and GPIOs corner names often contain a second voltage value. The name kept by the script could then differ from the database entry used to load the complete library set. The PVT values were valid, but the generated name was not the one expected by the rest of the flow, leading to an error when running the flow.

Two temporary solutions were possible. The generated corner file could be corrected manually, or the symbolic links in the technology library folder could be created in two passes. The first pass exposed only the standard-cell libraries while the corners were generated. The second restored the SRAMs, GPIOs, and other library views. Both solutions required the user to prepare the library folder around the behavior of the script.

The new generator obtains this information from a separate technology corner file instead. During technology integration, the user lists every corner with its exact database name and its process, voltage, and temperature values. The script reads only this small configuration block and no longer opens the Liberty files. It also checks for an empty corner list, repeated names, missing values, and values assigned to a corner that was not declared. The generated `corner_defs.tcl` therefore uses the same names as the technology database.

The complete corner list is kept in the generated file, but the user can choose which corners are active for a run. By default, all declared corners are enabled. This can instead be changed to disable first all corners, and then individual corners are enabled one by one. It is also possible to keep the default and disable only selected corners. If the final list is empty, the flow stops with an error. This selection supports the normal multi-corner run, a smaller subset, or a single-corner run without changing the technology libraries.

The existing setup and hold classification was preserved. The low-voltage corners at the temperature extremes are used as setup candidates, while the high-voltage corners at the temperature extremes are used as hold candidates. After the user selection is applied, only enabled corners remain in the corresponding analysis views.

A fallback is needed when the selected subset contains no corner for one of the two previously computed case. In that case, the generator uses all enabled corners for the empty role and prints a warning. Setup and hold are checked separately. For example, selecting only the two setup corners leaves the normal setup views unchanged, but there is no hold corner left. The same two enabled corners are then used for the hold views. If only one TT corner is selected instead, that corner is used for both setup and hold. The fallback does not claim that this corner is the worst case. It only keeps the selected run complete instead of generating an empty analysis view.

The technology-specific names are replaced here with the descriptive aliases `TT_corner`, `SS_corner_1`, `SS_corner_2`, `FF_corner_1`, and `FF_corner_2`. Table 3.3 shows how the selection and fallback behave with these names.

Table 3.3: Setup and hold views produced for representative corner selections.

Enabled corners	**Resulting analysis views**	**Fallback**
Complete set	Setup: `SS_corner_1`, `SS_corner_2`. Hold: `FF_corner_1`, `FF_corner_2`.	None
`TT_corner` set	Setup: `TT_corner`. Hold: `TT_corner`.	Setup and hold
`SS_corner_1`, `SS_corner_2` set	Setup: `SS_corner_1`, `SS_corner_2`. Hold: `SS_corner_1`, `SS_corner_2`.	Hold only

The new corner selection and its fallback behavior were checked through Level 1 using selected corner sets. The complete validation matrix and the resulting comparisons are reported in Section 3.8. The existing MMMC organization did not need to change. The flow now uses the declared corner names and can run with the number of corners needed for each case.

3.8 Regression Tests and Validation across Different Configurations

The changes presented in this chapter were tested with ESP release 2026.1.0 and the 12 nm technology setup described in Section 2.3. The PDK, libraries, and original corner names are not identified because of the signed non-disclosure agreements. RTL simulation used ModelSim Deluxe 2023.2, while synthesis used Cadence Genus 25.1. The tests were divided into two parts. The first checked the DCO simulation models. The second ran the revised constraints and corner selection through logic synthesis.

The common system was a 2×2 SoC with one IO tile, one Ariane processor tile, one `FFT_STRATUS_basic_fx32_dma64` accelerator tile, and one memory tile. Some tests synthesized every tile and the NoC, while smaller cases used only the IO tile or the standalone NoC domain socket. The clock periods used for synthesis are reported in Table 3.4.

Table 3.4: Clock periods used for the Chapter 3 synthesis tests.

Clock	Period	Clock	Period
`TILE_CLK`	1.7 ns	`NOC_CLK`	1.5 ns
`TEST_CLK`	40 ns	`FPGA_CLK`	20 ns
`IOLINK_CLK`	20 ns	`ERX_CLK`	40 ns
`ETX_CLK`	40 ns	`MDC_CLK`	1.6694 μs

The DCO tests are summarized in Table 3.5. Test 30A selected the behavioral model explicitly. Test 30B instead used the structural model and an SDF file generated with Genus. Test 30C left the model selection to the simulation flow. These three tests used the multi-DCO strategy. Test 30D then covered the dual external and single-DCO strategies. All simulations completed successfully.

Table 3.5: DCO simulation tests.

Test	Clock strategy	Simulation configuration	Pass
30A	Multi DCO	Behavioral DCO model selected explicitly	Yes
30B	Multi DCO	Structural DCO model with generated SDF	Yes
30C	Multi DCO	Automatic DCO model selection	Yes
30D	Dual external and single DCO	Default model selection for both strategies	Yes

These simulations check the behavior of the generated system, but they do not reach Level 1 by themselves because they do not all include logic synthesis. The configurations were then reused in the constraint tests, so the same three clock strategies were checked again through synthesis.

The completed constraint tests are shown in Table 3.6. Test 31A used the multi-DCO strategy and enabled all optional interfaces. It synthesized every tile and the NoC. Test 31B covered the same targets with dual external clocks and all optional interfaces disabled. Test 31C used the single-DCO strategy and synthesized the IO tile with the optional interfaces disabled. In every case, synthesis completed, a gate-level netlist was generated, and the timing reports contained no violations. These results meet Level 1 for the tested configurations.

Table 3.6: Level 1 validation of the conditional SDC constraints.

Test	Clock strategy	Synthesis targets	Optional interfaces	Pass
31A	Multi DCO	Every tile and NoC	All enabled	Yes
31B	Dual external	Every tile and NoC	All disabled	Yes
31C	Single DCO	IO tile	All disabled	Yes

The corner-selection tests reused the configuration of Test 31A but synthesized only the standalone NoC router. The router was selected as a small synthesizable design to check that the corner-selection mechanism worked through synthesis. These tests were not intended to evaluate a complete SoC or find a preferred corner set. Test 32A enabled the complete corner set and provided the baseline. Test 32B enabled only `TT_corner`, while Test 32C enabled `SS_corner_1` and `SS_corner_2`. All three runs completed Level 1. The resulting synthesis estimates are compared in Table 3.7.

Table 3.7: Synthesis results for the selected PVT corner sets.

Test and enabled corners	Area (μm^2)	Estimated power (mW)	NoC-router slack (ps)
32A: complete set	32810.121	30.51	69.2
32B: `TT_corner`	31586.199 (−3.7%)	30.51 (+0%)	150.0 (+116%)
32C: `SS_corner_1`, `SS_corner_2`	32792.167 (−0.05%)	30.54 (+0.09%)	80.0 (+15.6%)

The area value includes cell area, estimated net area, and macro area. Power was estimated with a constant activity factor of 0.2 and without an activity file. Therefore, these values are preliminary synthesis results, as expected from Level 1. They are useful for comparing these runs, but they should not be taken as a representative for power analysis.

The comparison also shows why the selected corner set must be stated together with the results. Using only the typical corner increased the reported slack and reduced the synthesized area relative to the complete set. This improvement is also expected because the TT timing model is more optimistic than a full MMMC sweep or an analysis limited to the SS corners. The two slow corners produced values much closer to the baseline. This does not mean that a smaller corner set improves the design. It means that synthesis is optimizing and checking the design under different conditions.

Overall, the tests cover all three clock strategies, both DCO models, different peripherals, and three PVT corner sets. The revised flow reaches Level 1 for these configurations, but not full configuration coverage or physical-implementation closure.

Chapter 4

Flow Modularization and Reuse for Standalone Designs

The ESP ASIC flow was developed for complete SoC configurations and their hierarchical blocks. Its implementation stages therefore rely on ESP-specific assumptions about the configuration, directory structure, supported top-level entities, constraints, and physical inputs. These assumptions make it difficult to apply the same methodology directly to an RTL design that is not generated as a standard ESP target.

This chapter investigates how far the implementation methodology can be separated from these platform-specific dependencies. A modified NoC router is used as a standalone case study. The aim is not to produce a universal flow or a tapeout-ready design, but to identify which stages can be reused and which inputs must remain under the control of the designer.

The chapter first examines the dependencies of the original flow and defines the inputs needed by a standalone design. It then describes the extracted `simpleflow` structure and evaluates it through synthesis and physical implementation of the router. The final section discusses the limits of the result and the possible extension toward a more general exploration flow.

4.1 Motivation and Scope

The ESP ASIC flow is built around the hierarchy and configuration of an ESP SoC. Although its hierarchical organization allows tiles and NoC routers to be implemented as separate blocks, these blocks are still prepared from a known system configuration. Their sources, constraints, working directories, and physical inputs follow the structure expected by the methodology. A standalone RTL design that does not belong to this hierarchy is therefore not a native implementation target of the flow.

Supporting this type of design can be useful for the Columbia system-level design (SLD) Group. A custom block may need early ASIC estimates before it is integrated into a complete system, or even before the decision to integrate it into ESP is made. Reusing the existing methodology would provide a common path for estimating timing, area, power, utilization, and routability. It could also complement FPGA prototyping when the same design needs to be evaluated in both implementation domains.

The purpose is not to replace the hierarchical ESP flow, but to make its main implementation stages available to designs outside its predefined block structure.

The case study originated from a request within the group to obtain power, performance, and area estimates for a modified look-ahead NoC router intended for machine-learning accelerators. The supplied top level was the SystemVerilog module `lookahead_router_top_flat`, derived from the router implementation already present in ESP.

In the normal platform hierarchy, this router is integrated through the ESP wrapper layers. In this experiment, the SystemVerilog module was instead used directly as the synthesis top level. The initial request required synthesis results for an ongoing research manuscript, while this thesis extended the experiment through physical design to evaluate the reuse of a larger part of the methodology.

A separate flow directory was created at the project root rather than modifying a generated ESP design directory. Most of the synthesis and physical design steps were kept in place, while the design sources and implementation inputs were loaded through a different path. This does not make the flow independent of all design information. In particular, the timing constraints must describe the behavior of the supplied block, and its floorplan and pin placement must follow its physical interface. The current implementation also retains fixed values for the selected technology and for some floorplan parameters, exactly like the original flow. The proof of concept therefore concerns the separation of the reusable implementation stages from these design-specific inputs, rather than automatic support for any arbitrary RTL design.

According to the validation levels defined in Table 3.1, the case study reached Level 2 physical-implementation validation for the reported design, constraints, technology setup, and pre-filler routed state. Routing and parasitic extraction were completed without reported setup or hold violations, while the available abstract-view design-rule and antenna checks passed. The connectivity run reported only the unconnected VDD terminals expected for a hierarchical block whose power-delivery connections are completed at the chip top level.

The unresolved filler-cell insertion prevented completion of the final GDSII layout, so Calibre DRC and LVS were not performed and Level 3 tapeout-readiness validation was not reached. Logic-equivalence checking and gate-level simulation were also outside the completed evaluation. The Level 2 result therefore demonstrates reuse through physical implementation under the reported conditions, but it does not establish functional equivalence or signoff completion.

4.2 ESP Design Flow Dependencies

The standalone experiment used ESP release 2026.1.0. The private ASIC flow does not follow the same release numbering and was taken from its main development branch at the time of the experiment. This difference is relevant because the two repositories do not define a single versioned environment. The design hierarchy comes from the public ESP release, while the scripts that prepare and implement its partitions evolve separately in the private repository.

The flow is hierarchical, but its hierarchy is not open to an arbitrary design. Its supported block-level targets are the tiles and the complete NoC domain socket generated for the selected SoC. A top-level target is also generated, but it follows a different path and includes assumptions about pads, GPIO libraries, and the integration of the block-level views. Therefore, using the raw router as the chip top level would not have represented the requested block-level experiment.

The main assumptions that prevented the router from entering the flow directly are summarized in Table 4.1.

The project preparation begins with `make esp-xconfig`, which selects the SoC hierarchy and its main parameters. The subsequent `make asic-setup` step creates the `ESP_SoC` working directory and one folder for every unique tile type, the NoC block, and the chip top level. It also populates the design-related entries of the ESP database. This

database is implemented as a collection of TCL arrays whose entries describe the known blocks and provide values consumed by later stages of the flow.

Table 4.1: Design assumptions of the hierarchical flow and their effect on the standalone router.

Dependency	**Assumption in the ESP flow**	**Effect on the standalone router**
Project preparation	Every implementation target is created from the selected ESP configuration	No working directory or design-database entry was generated for the supplied block
Design class	A target is a known tile, the NoC domain socket, or the chip top level	The raw router did not match a supported block type, while the chip-level path introduced unrelated integration requirements
Source loading	The file list, compilation commands, and top-level entity are obtained from the generated configuration	The SystemVerilog sources and their compilation order had to be specified separately
Timing setup	The NoC constraint and analysis files refer to the complete domain-socket hierarchy	The raw router interface was not described by the existing timing setup
Floorplanning	Dimensions are generated for known ESP partitions from their area and configured physical parameters	The standalone block required its own outline while preserving the technology-grid requirements
Pin placement	Port recognition and placement follow the interfaces of the supported ESP tiles and NoC block	The ports of the new top level were not recognized by the existing generator

This setup is more than directory creation. Conditions in the scripts select behavior from the block type and from the variables stored for that block. Since `lookahead_router_top_flat` was not part of the selected ESP configuration, `make asic-setup` had no information from which to create its working directory or database entries. Adding it as another name was also not sufficient because its hierarchy and ports did not match those of the supported NoC partition.

In the standard implementation, the NoC block is synthesized from `noc_domain_socket`. This entity contains the six NoC planes together with reset, configuration, and tile-interface logic. Each plane is instantiated through `sync_noc_set` and `sync_noc_xy`; the latter can add the asynchronous FIFOs that cross between `tile_clk` and `noc_clk`. The

VHDL entity `router` then wraps the SystemVerilog look-ahead router implementation. The public source tree therefore presents the router as one element of a larger mixed-language block [33].

The supplied RTL used a different boundary. Its top level, `lookahead_router_top_flat`, groups the router implementations for the NoC planes directly and does not include the surrounding ESP socket, clock-domain crossing, or configuration logic. A comparison with the public ESP router sources also showed that the supplied files are an ESP-derived variant rather than an unchanged copy of the public directory. Some base router files match, while the multicast, FIFO, arbiter, package, and wrapper sources contain differences. Because the exact ESP revision used as the starting point was not available, it was not possible to determine which differences were introduced specifically for this router variant.

Source loading was another consequence of the generated hierarchy. For the supported blocks, `make genus-setup` reads the file list also used for RTL simulation and writes the `read_hdl` commands consumed by Genus. The top-level entity is obtained from the design generated by `esp-xconfig`. Some later operations also refer directly to ports of the known ESP blocks.

The standalone design instead required an explicit source-loading sequence. The SystemVerilog package had to be read before the modules that use its types and parameters, followed by the lower-level router modules and finally `lookahead_router_top_flat`. Reading the top level before its package and dependencies prevented Genus from elaborating the design correctly. The existing setup generated this sequence only for designs registered in the ESP configuration database, so the commands for the standalone router had to be written separately.

The same boundary appeared in the timing setup. The standard `noc.sdc` file is written for `noc_domain_socket`, including the clocks and interfaces introduced by its wrapper hierarchy. It could not constrain the raw router top level because that design exposes a different clock and port interface. The original flow also loaded libraries and analysis views through its MMMC setup. No corresponding MMMC file was consumed for the logic synthesis of the standalone design, so the experiment required a separate constraint and library-loading path. The synthesis steps and optimization directives were kept as in the original flow.

Physical design had a similar division. The floorplan procedure of the hierarchical flow derives the dimensions of known ESP blocks from their area and target utilization. This was not sufficient for the standalone router because it did not account for the edge length required by its pins. The adapted pin-location script therefore computed minimum

width and height values together with the pin coordinates. Explicit dimensions selected from these bounds were then passed to the standard Innovus floorplanning command, while retaining the grid constraints required by the technology.

The pin-location generator was more closely tied to the design. It supports the known tile and NoC interfaces, identifies their signal groups, and places them according to the relative positions of routers and tiles. It is partly parameterized, but its parameters describe ESP blocks rather than a general list of arbitrary ports. The direct interfaces of `lookahead_router_top_flat` consequently needed a different mapping. The physical difficulty of that mapping, including the available edge capacity and grid-aligned coordinates, is evaluated later in the case study.

These dependencies were concentrated around the entry and physical interface of the design. Once the standalone RTL, constraints, libraries, floorplan, and pins were supplied, most synthesis and physical design stages could remain in place. This distinction motivates the requirements for standalone reuse discussed in the next section: the flow can reuse its implementation stages, but it still needs an explicit description of the design and its implementation intent.

4.3 Requirements for Standalone Design Reuse

The router experiment did not introduce a general configuration interface. Its sources, constraints, libraries, floorplan values, and pin-placement commands were inserted directly into the standalone scripts. This was sufficient for obtaining the requested implementation results, but it would require similar changes for every new block. The experience can nevertheless be used to identify the information that a future interface should expose.

This information belongs to three different parts of the methodology. Some inputs describe the design and cannot be obtained from the implementation flow alone. Others come from the selected technology and should be shared by every design using that technology. The synthesis and physical design stages form a third part: their main procedures can remain under the control of the flow once the first two groups of inputs are available.

Table 4.2 summarizes a possible organization. It does not describe a configuration file implemented during this case study. Instead, it defines the requirements derived from the adaptations that were necessary for the router.

Table 4.2: Proposed inputs for the reuse of the ASIC flow with a standalone design.

Input category	**Required information**	**Possible future mechanism**
Design definition	RTL sources, compilation order, top-level module, and parameter values	A source list and a standalone design configuration
Timing intent	Clocks, I/O constraints, timing exceptions, and applicable loads	A design-specific SDC supplied and checked by the design owner
Technology setup	Libraries, operating conditions, physical technology files, and extraction data	Selection from the technology configurations supported by the flow
Physical target	Aspect ratio, utilization objective, or explicit block dimensions	Configurable floorplan parameters with optional automatic sizing
Pin organization	Port groups, sides, ordering, layers, spacing, and grid requirements	A pin description translated into legal coordinates by a generator
Power information	Power-net conventions, implementation options, and representative switching activity when available	Flow settings together with an optional activity file or documented default assumptions
Flow controls	Selected stages, optimization effort, and report and output locations	A standalone run configuration

The design definition is the first input because the intended compilation unit cannot be inferred from a directory of RTL files. The designer must identify the top level, its parameter values, the required packages and modules, and their compilation order. A future flow could read this description and generate the tool commands that were written manually for the router. This would remove the dependency on the ESP-generated source list without making assumptions about the structure of the new block.

Timing intent requires a stronger division of responsibility. A template can provide locations for clock definitions, uncertainties, I/O delays, loads, and exceptions, while the flow can check that the file exists and report unconstrained paths. It cannot determine whether these constraints describe the intended operation of an unknown block. The design owner must therefore provide and validate the SDC. This is a necessary input rather than a limitation that can be removed through source parsing.

The physical inputs have a similar boundary, although more of their processing can be automated. Core dimensions may be supplied directly or estimated from the synthesized

area and a utilization objective. However, area alone may not produce a feasible outline when a block has many interface signals. The designer must also describe the meaning and desired organization of the ports. A reusable generator could then convert the requested sides, groups, and ordering into coordinates that follow the placement and routing grids of the selected technology. It could also report when the requested organization does not fit the available boundary.

Technology information should remain separate from these design choices. Library files, operating conditions, placement sites, routing layers, grids, and extraction data belong to the supported technology configuration and should not be repeated for every block. The standalone configuration would select among the available setups, while the flow would load the corresponding files. Power and ground conventions can follow the same arrangement.

Power estimation needs one additional optional input. When a representative VCD, SAIF, or equivalent activity file is available, it should be accepted by the flow. Otherwise, the configuration may define default switching assumptions, but these values and their use must be recorded in the reports. The resulting power value would then remain a preliminary estimate rather than a workload-based characterization. Power-oriented implementation options, such as clock gating, can still be exposed independently of the activity source.

Once these inputs are provided, the common implementation sequence does not need to know whether the design came from an ESP hierarchy or from a separate RTL project. Elaboration, synthesis, placement, clock-tree synthesis, routing, parasitic extraction, and report generation can use the same procedures, subject to the selected technology and tool options. A future standalone interface would therefore not replace the existing methodology. It would provide a controlled entry point to it. The next section describes the simpler, directly coded separation used for this proof of concept.

4.4 Extraction and Configuration of the Simplified Flow

The standalone implementation was organized in a directory named `simpleflow`, created at the root of the project. It was kept outside the working directories generated under `ESP_SoC`, since no entry for the router existed in the ESP design database. The directory was also not created through a new Make target. Instead, it provided a

separate location for the design inputs, tool scripts, intermediate files, and reports needed by this experiment.

The main directories used by the implementation are shown below. Temporary copies created while preparing the technology files are omitted.

```
simpleflow/
|-- constraints/
|-- ext/
|-- lef/
|-- lib/
|-- rtl/
|   |-- router/
|   `-- tb/
|-- synthesis/
|   |-- tcl/
|   |-- outputs/
|   `-- reports/
|-- pnr/
|   |-- tcl/
|   |-- import_design/
|   |-- reports/
|   `-- timingReports/
`-- signoff/
```

The `rtl` and `constraints` directories separated the standalone design from the generated ESP sources. The `lib` and `lef` directories contained only the views selected for this implementation, rather than all the libraries available in the project technology directory. Synthesis and physical design then had their own TCL, report, and output locations. This structure resembles the separation already used by the hierarchical methodology, but it no longer requires a generated tile, NoC-domain, or top-level working folder. The commands were first tested individually in the Genus and Innovus batch or command-line environments. After the sequence worked, they were consolidated into TCL scripts inside `simpleflow`. This was a direct way to complete the requested implementation, but it also meant that paths, design names, and several options were written for this case rather than obtained from a separate configuration file.

For synthesis, the ESP-generated source-list preparation was replaced by explicit `read_hdl` commands. The SystemVerilog package was loaded first, followed by the router modules in their dependency order and finally `lookahead_router_top_flat`. The standalone top-level name and its parameter values were selected in the same

scripts. Genus then elaborated this hierarchy without relying on the top level generated by `esp-xconfig`.

The timing and library setup was simplified as well. Logic synthesis loaded one selected Liberty file directly and read one design-specific SDC; it did not consume the generated MMMC setup of the hierarchical flow. Once these inputs had been loaded, the existing synthesis, optimization, clock-gating, and report-generation sequence could be retained. The selected timing target, constraints, library choice, and power assumptions are reported with the case-study setup rather than treated as general properties of `simpleflow`. The main synthesis output passed to physical design was the gate-level netlist. Innovus still required an MMMC description for timing analysis, even though the experiment used only one view. A temporary MMMC TCL file was therefore created with one library set containing the selected Liberty file, one constraint mode based on the standalone SDC, one timing corner, and one analysis view, valid for both setup and hold STA paths. This setup should be distinguished from the synthesis path, where the library and SDC were loaded directly. It reduced the analysis space to one mode and one corner, but it did not remove the timing-view data model expected by Innovus.

Floorplan initialization and pin placement required separate treatment. The utilization-based floorplan of the hierarchical flow could produce an outline large enough for the synthesized cells while still leaving insufficient space for the router interface. For this reason, the adapted pin-location script did more than generate pin coordinates. It organized the directional and local interfaces, aligned their locations with the allowed routing and placement grids, checked for overlap, and reported the minimum block width and height needed to accommodate them. The floorplan itself was created with the standard Innovus command, not by a new Python floorplan generator. Explicit width and height values were selected from the minimum bounds reported by the pin script and written into the physical design TCL. A target-utilization value was not used to derive these dimensions because it did not represent the boundary-pin requirement. After the floorplan was initialized, the pin generator produced the TCL commands for the individual pin locations, and this file was sourced by Innovus.

With the design imported and these initial physical inputs in place, the existing sequence could continue through placement, physical optimization, clock-tree synthesis, routing, post-route optimization, parasitic extraction, and report generation. The corresponding TCL procedures required much less adaptation than the design-loading and floorplanning steps.

This sequence formed the main reusable core of the experiment.

Not every directory represented a completed flow stage. The `sim` directory and the formal-verification area under `synthesis/fv` were not used, since gate-level simulation and equivalence checking were outside the completed experiment. The `signoff` directory was used for parasitic extraction and for the filler-cell insertion attempt. Its presence therefore does not indicate that physical signoff was completed. The filler issue and its effect on the final validation boundary are discussed with the physical design results.

Overall, `simpleflow` provided a working path from RTL to an abstract view-based physical implementation. It extracted the main synthesis and physical design sequence from the ESP hierarchy, but the new entry points were still configured directly for the router. The source commands, timing setup, floorplan dimensions, and pin interpretation would need to change for another design. In this sense, the directory demonstrates the feasibility of reuse, while the requirements defined in the previous section describe how the same approach could later become a general interface.

4.5 Standalone NoC Router Case Study

4.5.1 Router Configuration and Experimental Setup

The case study used the same proprietary 12 nm technology setup described in Section 2.3. The PDK and library vendor are not identified because of the signed non-disclosure agreements. The results therefore identify the technology node and the selected threshold-voltage flavor, but not the original technology, library names, nominal voltage, or nominal temperature.

The implemented top level was `lookahead_router_top_flat`. It directly contains one router for each of the six ESP NoC planes and exposes the North, South, East, West, and Local interfaces of every plane. Planes 1–3 use a 64-bit payload and plane 5 uses a 32-bit payload. The two-bit packet preamble increases their physical data-port widths to 66 and 34 bits, respectively. Planes 4 and 6 were configured with `DMADataWidth` equal to 256, resulting in a 258-bit data port after the same preamble was added.

The 256-bit configuration intentionally selected the multicast router implementation for planes 4 and 6. A preliminary 64-bit configuration used the non-multicast branch, but it was evaluated only during the first synthesis experiments. Therefore, the physical implementation and the results reported in this section refer only to the 256-bit multicast configuration; they are not presented as a direct width-scaling comparison. The default queue depth of four and all five ports were retained.

This top level is different from `noc_domain_socket`. It contains the six router instances, but not the socket configuration logic, reset synchronizers, or asynchronous FIFOs used at the tile-to-NoC clock-domain boundaries. All router instances are instead driven by the same top-level clock.

The implementation used Cadence Genus 25.1 and Innovus 25.1. One LVT standard-cell library was selected at the TT corner, with nominal voltage and temperature. LVT cells were chosen to support the 1 GHz timing objective; this was an implementation choice rather than a comparison against other threshold-voltage flavors. The given objectives also requested a utilization above 70% and power below 40 mW. These values were initial targets for the experiment, not a complete tapeout specification.

The large interface became an additional physical requirement. With the 256-bit setting, the flattened top level contains 7,610 pins. This did not prevent synthesis, but it constrained the minimum floorplan perimeter and eventually made the requested utilization incompatible with the required pin organization.

4.5.2 Timing Constraints and Logic-Synthesis Results

The standalone SDC was written from the interface of `lookahead_router_top_flat`, using the ESP NoC constraints as a reference. It creates one clock with a period of 1 ns and an explicit 50% duty cycle. All functional input ports except the clock and reset receive input-delay constraints, while all output ports receive output-delay constraints. The maximum I/O delay is 20% of the clock period, or 200 ps, and the minimum delay is −5% of the period, or −50 ps.

Clock uncertainty, clock transition, the driving cell, output load, maximum transition, and maximum fanout are read from the technology constraint file. Their values are not reported because they contain information tied to the confidential library setup. The reset is excluded from the general input collection and receives a false-path constraint because it is asynchronous. This SDC therefore replaces the constraints written for the complete NoC-domain socket while retaining the same technology-level assumptions used by the original flow.

Genus loaded the LVT Liberty file and this SDC directly. Clock gating was enabled to reduce unnecessary clock activity. Since no simulation-derived activity file was supplied, input and sequential switching activities were both asserted as 0.1 for power analysis. The estimate is therefore based on explicit activity assumptions, rather than on the traffic produced by a representative router workload.

Synthesis completed without reported negative setup slack at the 1 GHz target. The mapped design contained 54,323 cells and occupied 23,119.230 μm^2 of standard-cell area. The activity-asserted power estimate was 26.53 mW, divided into 8.89 mW of switching power, 17.46 mW of internal power, and 0.1693 mW of leakage power. These results provided the initial PPA values requested for the router and the gate-level netlist used by the physical design flow. They are compared with the post-route values in Table 4.4.

4.5.3 Floorplanning, Physical Implementation, and Results

The pin generator classified the flattened ports by plane and direction. The North, South, West, and East interfaces were assigned to the corresponding top, bottom, left, and right block edges. The Local interface and the global control pins were divided between the top and left edges. Within each region, the script preserved a deterministic signal order and distributed the pins across routing layers whose preferred direction is perpendicular to the selected edge. The resulting coordinates were snapped to the legal technology grid before the placement commands were written.

Each cardinal interface contains 1,520 pins. The top edge contains the North interface together with 770 Local and control pins, for a total of 2,290. The left edge similarly contains the West interface and 760 Local pins, for a total of 2,280. The bottom and right edges contain 1,520 pins each. The generator reported the required edge dimensions shown in Table 4.3.

Table 4.3: Pin distribution and minimum side lengths reported by the standalone pin generator.

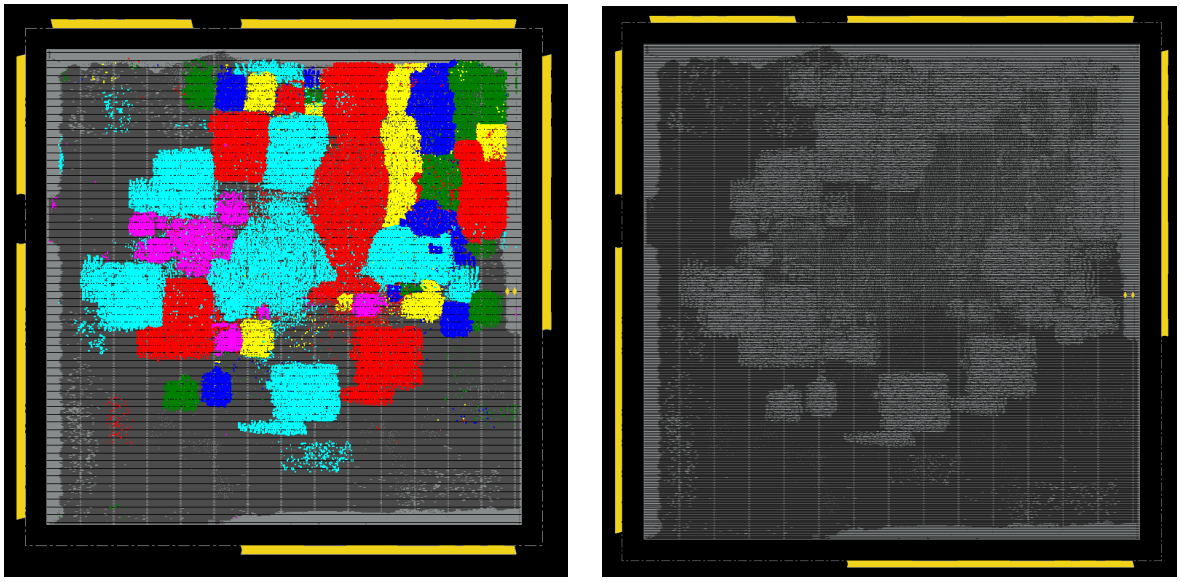
Edge	Pins	Regions	Minimum side length (μm)
Top	2,290	2	336.608
Bottom	1,520	1	234.304
Left	2,280	2	335.328
Right	1,520	1	234.304

A square 340 μm by 340 μm core was selected from these bounds. It is only slightly larger than the minimum dimensions imposed by the top and left edges, but it gives every pin group enough legal space. A 15 μm margin was added on each side, producing a die area of 370 μm by 370 μm. These margins separate the core from the die boundary; they do not contain GPIO pads or a chip-level I/O ring.

The pin requirement explains why the original utilization objective could not be retained. A floorplan derived only from the synthesized cell area and a target above 70% would

have been too small for the interface. The explicit core instead allowed the pin TCL file to be applied without overlap or grid violations. Figure 4.1a shows how the six router instances were distributed after placement.

After floorplanning, the reused stages completed placement, clock-tree synthesis, routing, post-route optimization, and parasitic extraction. The routed design reported no setup or hold violations in the selected single analysis view. Figure 4.1b shows the physical design view before final filler insertion. The yellow structures at the boundary are the dense groups of router pins.



(a) Placed instances highlighted by plane. **(b)** Routed view before final filler insertion.

Figure 4.1: Innovus physical design views of the 256-bit multicast router. In a, plane 1 is green, plane 2 is blue, plane 3 is yellow, plane 4 is red, plane 5 is magenta, and plane 6 is cyan.

The PPA results at the two implementation stages are summarized in Table 4.4. A zero value for WNS means that no negative slack was reported; it does not indicate that the available positive margin was exactly zero. Hold timing was not included in the post-synthesis result.

The increase in cell area and count includes the cells added during physical implementation, including the clock tree and physical optimization. The final cell/core utilization, computed as the standard-cell area divided by the core area, was 30.94%. The low utilization is therefore a consequence of the pin-limited outline, not a failure to place the logic.

Post-route power was estimated with the same activity-asserted approach used for synthesis. The reported total increased to 51.98 mW, with 26.77 mW of switching power, 24.84 mW of internal power, and 0.3672 mW of leakage. The physical netlist contains

additional cells and the post-route analysis includes routed capacitance and extracted parasitics. At the same time, the asserted activities do not model how the combinational paths of the router are active under real traffic. The power values are therefore useful as preliminary implementation estimates, but they cannot establish if the router satisfies the requested 40 mW objective for a representative workload.

Table 4.4: Post-synthesis and post-route results for the 256-bit multicast router configuration.

Metric	**Post-synthesis**	**Post-route**	**Information**
Setup WNS (ps)	0	0	1 ns clock; no negative slack
Setup TNS (ps)	0	0	
Hold WNS (ps)	N/A	0	Not reported after synthesis
Hold TNS (ps)	N/A	0	
Standard-cell area (μm^2)	23,119.230	35,772.565	Post-route includes additional cells
Cell count	54,323	80,229	Includes clock-tree and physical cells
Core area (μm^2)	N/A	115,600	340 μm × 340 μm size
Die area (μm^2)	N/A	136,900	370 μm × 370 μm size; no pad ring
Cell/core utilization (%)	N/A	30.94	Standard-cell area over core area
Total power (mW)	26.53	51.98	Activity asserted at 0.1; no activity file
Switching power (mW)	8.89	26.77	Same activity assumptions
Internal power (mW)	17.46	24.84	
Leakage power (mW)	0.1693	0.3672	LVT library used

The available abstract-view checks completed with zero design-rule violations and zero antenna violations. The block-level connectivity procedure reported the expected unconnected VDD terminals because the power-delivery network of hierarchical blocks is completed at the chip top level. This is the standard result for the block-finishing stage of the established flow and was not treated as a functional connectivity failure.

Final filler insertion remained unresolved. The standard hierarchical flow uses mixed libraries and includes a one-site filler, while the selected single-library setup had a different minimum filler size rule to be applid. The existing filler sequence therefore left gaps that prevented completion of the final layout. Using a filler compatible with the available gap sizes was identified as a possible correction, but it was not tested in this experiment.

Despite this final issue, the case study completed the stages required for Level 2 validation: synthesis, placement, clock-tree synthesis, routing, post-route timing closure, and parasitic extraction. The filler limitation prevented final GDSII generation and the subsequent Calibre DRC and LVS checks, which remain part of Level 3 rather than the physical-implementation claim made here.

4.6 Discussion and Limitations

The case study shows that the ESP ASIC implementation methodology can be reused outside the standard ESP hierarchy. After loading the router RTL, constraints, and technology inputs, the established synthesis and physical-design stages ran without changes to their main sequence. This does not make the extracted flow general-purpose, but it shows that the methodology is not limited to blocks generated by an ESP configuration.

The work also identified the boundary of this reuse. Tool invocation and the main implementation stages can remain common, while the RTL source order, top-level module, timing constraints, library selection, floorplan, and pin placement still depend on the design. An SDC file for an unknown block cannot be generated safely without information about its clocks and timing relationships. A more general flow should accept these inputs through a stable interface instead of trying to infer them.

The physical results exposed another important limit. The initial utilization objective above 70% was based on the logic area, but the router outline was instead constrained by its 7,610 pins and their directional placement. The final cell/core utilization of 30.94% follows from the minimum legal floorplan rather than from ineffective placement. For blocks with very wide interfaces, floorplan dimensions and target utilization cannot be treated as independent requirements.

The power results are also preliminary. The activity-asserted model used a fixed activity of 0.1 and did not represent traffic through the router. It was sufficient to exercise power analysis, but it cannot determine whether the 40 mW objective is met under a realistic workload. A future evaluation would need simulation-derived activity from representative traffic.

The experiment reached Level 2 validation, including routing, timing closure, and parasitic extraction, but it did not reach tapeout readiness. The unresolved filler insertion prevented the final GDSII and Calibre checks, while formal equivalence and gate-level simulation were not performed. These limits restrict the conclusions to the tested router configuration, technology setup, and pre-filler routed design.

A next step is to turn the `simpleflow` directory into a reusable entry point that preserves the common stages while each custom block supplies its RTL, constraints, and physical-interface description. A more complex design, such as the Vortex GPU considered for ESP, could test this separation further and support a comparison between FPGA and ASIC implementations. The router case study is therefore a starting point for a broader exploration flow, rather than its final form.

Chapter 5

Technology Abstraction for Portable Logic Synthesis

Moving an ASIC flow to another technology requires more than replacing a standard-cell library. It must associate the required timing, simulation, physical, and extraction views with the cells, layers, supplies, design rules, and operating corners expected by its scripts. ESP stores this information in a technology database read during project setup and implementation. When this work began, the existing 12 nm configuration was the only complete example. A new port therefore required reconstructing an interface spread across configuration files, documentation, and the scripts that consumed them.

This chapter addresses that problem through a common technology-configuration interface. A fixed YAML template defines the database structure, a local manifest supplies the values and collateral references for one technology, and a Python generator produces the TCL files already read by the EDA flow. This arrangement keeps sensitive values outside the repository and leaves the downstream interface unchanged. The template covers the complete technology database, although the new 28 nm configuration is evaluated here only through logic synthesis. The work addresses RQ3 by separating the structure shared by every technology from the values that belong to one technology.

The port also exposed a separate design limitation. The target SoC uses the 32-bit Ibex processor, while the ASIC preparation path still contained memory assumptions inherited from the 64-bit Ariane configuration. After these scripts were updated, the completed path was applied to the 2×2 Fermilab case study. The CPU, IO, and SLM tiles and the per-tile NoC domain socket were synthesized to obtain preliminary area, timing, and assumption-based power information. The eFPGA and chip-level IO cells are considered separately because their estimates come from physical abstracts.

5.1 Motivation and Scope

This work originated from a collaboration between Fermilab and the System-Level Design Group at Columbia University on the implementation of an ESP-based SoC. Fermilab is responsible for the physical design, while the Columbia group provides the configurable ESP RTL and the associated front-end implementation support. Within the Columbia effort, the project supervisor is responsible for the RTL handoff and the planned gate-level netlist simulation. As the current active maintainer of the ESP ASIC flow, the author instead focused on integrating a 28 nm technology and producing preliminary area estimates for the supported tiles.

The target system is a 2×2 configuration containing an Ibex processor tile, a shared-local-memory tile with 512 KiB, an I/O tile, and a custom tile containing a proprietary embedded field-programmable gate array (eFPGA). Supporting the new technology required a complete set of technology-specific values for the database consumed by the implementation scripts. The work therefore introduced a fixed YAML template, a local technology manifest, and automated generation of the corresponding TCL database. The template defines a common structure for every technology, whereas each manifest supplies the collateral references and parameter values of its target technology. The resulting configuration is intended to support logic synthesis and the estimation of standard-cell and memory-macro area for the regular ESP tiles.

The established procedure for using an integrated technology repository within the ESP methodology is documented in Appendix A of the methodology guide [13, Appendix A]. That procedure assumes that the required technology metadata have already been populated and is therefore not repeated here. This chapter instead concerns the configuration interface that prepares those inputs and its application to the Fermilab case study. The evaluation is limited to logic synthesis; physical implementation and signoff of the 28 nm technology remain outside its scope. The custom eFPGA tile is also excluded because the current hierarchical flow supports its registered ESP targets rather than arbitrary custom blocks, a design boundary examined in Section 4.1. Its physical footprint may be considered separately by the physical design team, but it is not part of the synthesis-based estimates reported in this chapter.

5.2 Limitations of Existing Technology Integration

The ESP ASIC methodology keeps the information for each supported technology in a dedicated directory. In addition to the memory, pad, and DCO wrappers used to

connect technology-dependent components to ESP, this directory exposes the views required by simulation and implementation. Liberty timing libraries, behavioral Verilog models, LEF abstracts, transistor-level netlists, AOCV data, extraction files, and GDSII layouts are organized under directories such as `lib`, `verilog`, `lef`, `cdl`, `aocv`, `ext`, and `gds`. These directories normally contain symbolic links to the installed technology collateral rather than copies of the original files. The complete organization and naming convention are documented in the methodology database guide [13, Appendix B, Sec. B.1].

Placing the views in this structure is only one part of the integration. The `config` directory must also describe how the flow uses them. Its files assign technology metadata to elements of the `TECH` array, a TCL structure that acts as the technology database consumed by the Python generators and the EDA-tool scripts. The entries cover cells, design rules, file lists, routing layers, and supply pins. The `<tech>_corners.tcl` file forms an additional group introduced with the configurable PVT-corner support described in Section 3.7. Together, these files provide information needed across the flow, from library and constraint setup during synthesis to placement, routing, power planning, extraction, and signoff preparation.

The main limitation was not the directory organization itself, but the way a new configuration had to be created. At the beginning of this work, the 12 nm technology was the only complete technology supported by the flow and therefore the only working reference for another port. A partial template existed, but it did not provide an authoritative definition of the complete metadata interface used by the final configuration files. Integrating another technology consequently required comparing the template with the 12 nm configuration, the metadata documented in Appendix B, and the scripts that read individual `TECH(...)` entries. Without a complete template, the maintainer therefore had to determine the required metadata by examining the existing 12 nm configuration and the flow scripts.

The collateral links had a similar limitation. The shell script supplied for the 12 nm setup encoded the paths and naming conventions of that technology and could not be reused directly for another vendor's libraries and IPs. A universal linking script is difficult to define because vendors do not follow one naming scheme for cells, memories, I/O libraries, or their different views. As a result, an already configured technology could be reused across projects, but preparing a technology for the first time still required substantial manual reconstruction of both its metadata and its collateral mapping. This gap motivated the dedicated technology-configuration repository presented in the next section.

5.3 ESP Technology Configuration Repository

The new ESP Tech Config repository replaces the incomplete technology template with a common configuration interface. It is placed directly in the `config` directory of a technology repository, as shown in Figure 5.1. This keeps the generated database beside the technology wrappers and linked library views expected by the existing flow, without changing how those other parts of the technology directory are organized. In particular, creation of the symbolic links remains a technology-specific task because the names and directory layouts of the vendor collateral cannot be inferred reliably by a general-purpose script.

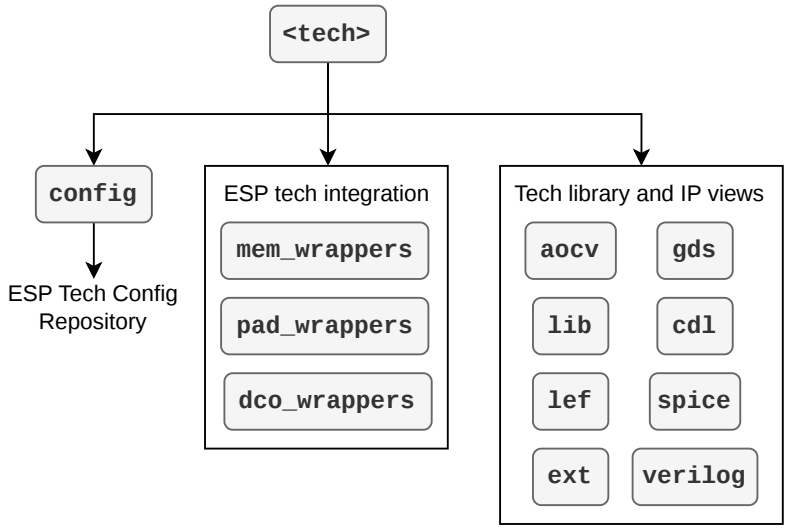


Figure 5.1: Position of the ESP Tech Config repository within a technology directory. The repository occupies the `config` branch, while the ESP wrappers and linked technology views remain separate.

The repository contains a Makefile, a fixed YAML template, the database generator, the manifest-update utility, and a `.gitignore` file. The template defines the complete structure of the `TECH` database, including the entries documented in Appendix B of the methodology guide [13, Appendix B] and the corner metadata introduced in Section 3.7. Entries that are not yet implemented by the flow are retained in the template as placeholders. A developer therefore starts from the same database structure for every technology and supplies only the values and collateral references appropriate to the technology being integrated.

Running `make manifest` creates the local `tech.yaml` manifest from this template. The command does not overwrite a manifest that already exists, so values entered during the port are preserved. The manifest is excluded from version control by default because it may contain installation paths or other sensitive technology-specific information that

should remain local. The generated TCL files are also excluded because they can be recreated from the repository files and the completed manifest. After the manifest has been filled in, `make tech-config` runs the Python generator and creates the six TCL configuration files.

Figure 5.2 summarizes the resulting workflow. The dashed line distinguishes the files present in a clone from those produced locally. Conceptually, the manifest holds one hierarchical description of the technology and is unrolled into six TCL database files. The generator is the program that performs this transformation: it reads the populated manifest, maps its six top-level metadata groups to the corresponding output files, and writes the `TECH(...)` assignments required by the flow. The generated files can then be sourced by the existing EDA scripts without manual editing or changes to their interface.

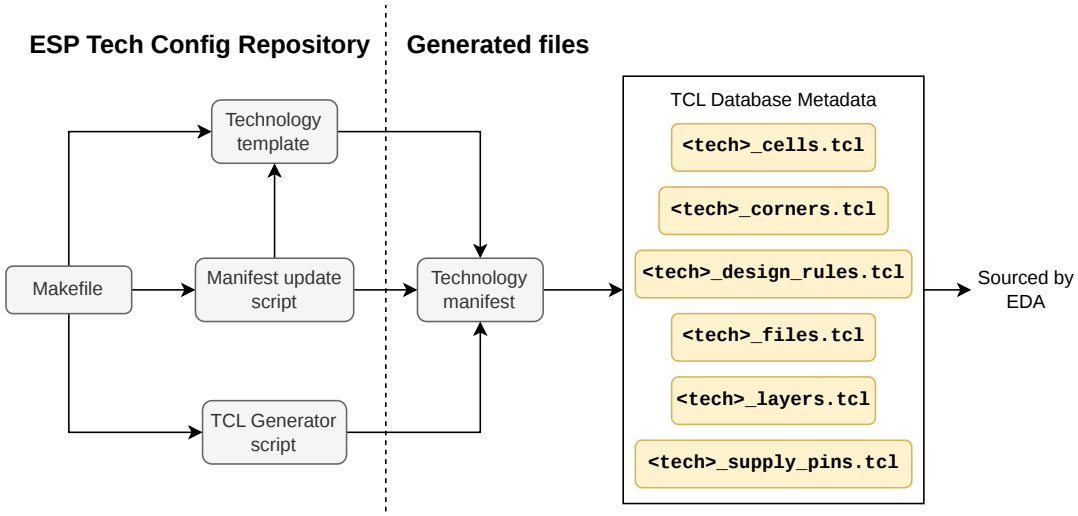


Figure 5.2: Conceptual workflow of the ESP Tech Config repository. The generator unrolls the local technology manifest into the six TCL files shown in yellow, which are then sourced by the existing EDA flow.

The six outputs describe cells, corners, design rules, files, layers, and supply pins. During generation, the template supplies the expected structure and the manifest supplies the technology values. For each field in the manifest, the generator writes a TCL assignment only when a usable value has been provided. Blank fields and placeholders such as `N/A`, `TBD`, or `TODO` are simply left out of the generated files. Nested YAML keys are flattened into the form used by the `TECH` array, while file entries receive the path handling needed by the existing technology-directory convention. This allows the template to cover the entire database, including metadata used after synthesis, without forcing an incomplete technology port to define unsupported values.

This generation step is also connected to the main ESP EDA flow. When `make asic-setup` is run to create the `ESP_SoC` working directory, the ESP EDA Makefile checks whether the selected technology's `config` directory provides the `tech-config` Make target. If it does, the setup calls `make tech-config` and uses the generated TCL files. If the target is not available, the setup still supports the previous configuration format: it continues with the existing TCL files and prints a warning that the legacy method is being used. Existing technology repositories therefore remain usable while new repositories can generate their configuration automatically.

The repository also provides `make manifest-update` as a maintenance convenience. This feature was introduced when the template changed after an initial 28 nm manifest had already been populated. It adds newly introduced template entries while preserving the values and comments already written in the local manifest, and reports local entries that no longer appear in the template. It is therefore useful when the common schema evolves, but it is not part of the normal generation path once a manifest is up to date.

Both Python utilities were checked with dedicated test scripts. The generated database was also tested through the completed synthesis runs for the CPU, shared-local-memory, IO, and NoC domain socket blocks discussed later in this chapter. These runs confirm that the generated TCL files are accepted by the synthesis flow for the case study. Although the template covers the metadata required by the complete ASIC flow, validation of the 28 nm configuration beyond synthesis remains outside the scope of this work.

5.4 Pre-Synthesis Integration of Ibex

Selecting Ibex in the ESP SoC Generator was not sufficient to prepare the design for ASIC synthesis. The RTL already supported the processor, but the memory-integration scripts still contained two assumptions inherited from the Ariane configuration. First, the memory-wrapper generator always expected an Ariane L1 memory. Second, the shared-local-memory description was always generated with 64-bit words. Both assumptions had to be removed before the wrappers for the Fermilab configuration could be generated.

The first problem occurred in `asic_memgen.py`. The script called the L1-wrapper generator unconditionally and expected the memory list to contain a 256×64 entry. This entry is required by the Ariane integration, but it is not part of an Ibex configuration. The Make target was therefore changed to pass the selected processor architecture to the script. The script checks that the value is one of the supported choices (Ariane, Ibex, or Leon3) and generates the L1 wrapper only when Ariane is selected. An Ibex

design no longer needs an unused L1 entry merely to complete the ASIC preparation step.

The processor selection and its word width remain separate configuration values. The generated `.esp_config` file identifies the processor with `CPU_ARCH = ibex`, while the SoC Generator derives `ARCH_BITS = 32`. Selecting Ariane instead sets `ARCH_BITS = 64`. Keeping this distinction is important because the L1 decision depends on the selected processor, while the SLM organization depends on the processor word width.

Previously, `socmap_gen.py` always described the SLM as a 64-bit-wide memory. The revised generator takes the word width from `ARCH_BITS` and calculates the required number of words from the configured capacity:

$$N_{\text{words}} = \frac{S_{\text{SLM}} \times 1024 \times 8}{W_{\text{arch}}},$$

where S_{SLM} is expressed in KiB and W_{arch} is either 32 or 64 bits. The result must be a positive power of two so that its address width can be obtained directly. For the 512 KiB SLM used in this work, the logical memory seen by the ESP RTL contains 131,072 32-bit words and has a 17-bit word address. A physical SRAM macro does not necessarily have the same depth or address width. The SLM memory generator bridges this difference by taking the logical requirement produced by `socmap_gen.py` and combining technology-specific memory wrappers until the requested depth and word width are obtained.

Supporting the Ibex configuration therefore required every stage around this generator to accept the 32-bit organization. The SLM policy in `asic_memgen.py` was extended to generate technology-specific wrappers for both 32-bit and 64-bit physical memories. At the other end of the path, `memory_asic.vhd` was extended with 32-bit SLM components and now selects the generated logical memory using both its address width and its data width. Unsupported combinations are rejected explicitly.

Together, these changes make the memory-generation path aware of the selected SoC configuration. Although they were motivated by the Ibex-based SoC, the SLM changes are based on the architectural word width rather than on the processor name and therefore also retain support for other 32-bit configurations. The existing 64-bit Ariane path is preserved. The physical SRAM selected for the 28 nm port, its wrapper validation, and the resulting tile implementation are discussed in the following section.

5.5 Fermilab SoC Case Study

This case study applied the 28 nm technology configuration and the Ibex changes to the SoC prepared for the Fermilab collaboration. The target system is intended for the evaluation of high-energy-physics algorithms and for regression tests run in the eFPGA. Only the information needed to describe the implementation is reported here; details of the intended algorithms and of the proprietary eFPGA remain outside the scope of the thesis.

5.5.1 SoC Configuration and Synthesis Setup

The SoC was configured as a 2×2 ESP system with a 25 mm^2 area budget. It contained an Ibex CPU tile, an IO tile, and an SLM tile with 512 KiB of storage. The fourth position was left empty in the generated configuration because it is reserved for a modified accelerator tile containing the eFPGA. This custom tile is not supported by the current hierarchical ASIC flow and was therefore not synthesized. Its physical footprint is considered separately at the end of this section. The 25 mm^2 value was a planning target, not a completed die-area measurement.

The IO tile used the I/O Link as the intended external communication interface. The UART was also present because the current ESP configuration does not allow it to be disabled, but it was not used by the case study. Other optional IO-tile peripherals were disabled.

The experiments used ESP release 2026.1.0 and Cadence Genus 23.1. The default MMMC flow was retained with all six corners enabled. Their original names and operating conditions are omitted because they belong to the confidential technology setup. Following the naming convention used in Section 3.7, the corners are identified as `TT_corner`, `SS_corner_1`, `SS_corner_2`, `FF_corner_1`, `FF_corner_2`, and `FF_corner_3`.

The SoC used the multi-DCO clock strategy. The clock periods are listed in Table 5.1. The 4 ns external clock constraint was retained as a fallback, but that clock did not drive the tiles or the NoC in this configuration.

The DCOs followed the structural soft-macro approach described in Section 3.4. Their generic structures were mapped to standard cells, then preserved and uniquified during synthesis. Their area is therefore part of the reported standard-cell area rather than the hard-macro area.

Table 5.1: Clock periods used for the Fermilab SoC synthesis runs.

Clock	Period (ns)	Role
`NOC_CLK`	2.0	NoC domain
`TILE_CLK`	2.5	Tile domains
`IOLINK_CLK`	20.0	I/O Link interface
External fallback	4.0	Inactive in the selected strategy

The normal ESP synthesis sequence was used, with one change to the technology-specific `dont_use` list. The inherited 12 nm flow contained restrictions for cells in its proprietary libraries. Because these restrictions were specific to that technology and their original rationale was not documented, they were not transferred to the new technology. For the 28 nm runs, only scan flip-flops were excluded. Genus could use all other available standard cells.

5.5.2 Memory Integration and Validation

The two memory systems required different compositions of the available SRAM macros. For the SLM tile, the 32-bit organization derived in Section 5.4 was implemented with 1024×32 physical SRAMs. Each instance stores 4 KiB, so 128 instances were needed to provide the configured 512 KiB scratchpad. The SLM generator created the banking and wrapper hierarchy needed to expose these instances as the single logical memory expected by the ESP RTL.

The IO tile required a 128 KiB boot memory organized through 2048×8 logical wrappers. The available physical SRAM was instead 1024×8. Each logical wrapper was therefore built from two physical instances. The most significant address bit selects which SRAM is accessed. The same selection controls the chip-enable signals and the output path, so that only the selected memory is active and drives the wrapper output. Each wrapper stores 2 KiB; 64 wrappers, and thus 128 physical SRAMs, were required for the complete boot memory.

The generated SLM and boot-memory wrappers were checked with dedicated testbenches before the tile synthesis runs. These simulations verified that the physical memories could be accessed through the logical interfaces expected by ESP. Full SoC RTL simulation is not discussed in this chapter, and gate-level netlist simulation had not been performed at the time of writing.

5.5.3 Synthesis Results and Additional Area Estimates

The IO tile, CPU tile, SLM tile, and NoC domain socket were synthesized separately. Table 5.2 reports the values from the final Genus `report_area` output together with setup WNS and TNS. The standard-cell area includes mapped logic such as the structural DCOs, while the macro area contains the SRAM instances. A zero slack entry means that no negative slack was reported. Each row describes one instance of the corresponding synthesis target.

Table 5.2: Logic-synthesis area and setup-timing results for the Fermilab SoC partitions.

Block	Std.-cell area	Macro area (μm^2)	Total area	Cell count	WNS (ps)	TNS
NoC domain socket (per tile)	33,843.474	0	33,843.474	27,976	0	0
IO tile	43,476.048	399,332.336	442,808.384	24,894	0	0
CPU tile	31,274.460	0	31,274.460	19,564	−315	−315
SLM tile	33,355.854	1,155,297.469	1,188,653.323	26,017	0	0

The NoC domain socket and CPU tile contained only mapped standard-cell logic. In contrast, SRAMs accounted for approximately 90% of the IO-tile area and 97% of the SLM-tile area. The NoC domain socket is instantiated once for every tile. Its table entry is therefore a per-tile value, not the complete NoC contribution. The four tile positions in this 2×2 SoC require four socket instances, for a combined socket area of 135,373.896 μm^2, or approximately 0.135 mm^2. The three synthesized tiles and these four sockets together occupy approximately 1.80 mm^2, of which about 1.55 mm^2 comes from SRAM macros. This total does not include the internal logic or eFPGA of the custom tile.

The CPU tile had one violating clock-to-output path. It ended at `cpuerr`, an Ibex monitoring signal, with both WNS and TNS equal to −315 ps. No other negative paths were reported. The mapped area remains useful for the preliminary estimate, but the CPU result is not presented as timing closed. The isolated output path should be reviewed in the Ibex RTL and its interface constraints before the netlist is used by later implementation stages.

Power was estimated at `SS_corner_1`. No simulation-derived activity file was available, so the analysis used a static probability of 0.5 and a toggle activity of 0.2. Table 5.3 reports the resulting total power and the fraction attributed to leakage.

Table 5.3: Assumption-based synthesis power estimates at `SS_corner_1`.

Block	Total power (mW)	Leakage (%)
NoC domain socket (per tile)	18.85	26.68
IO tile	19.54	47.20
CPU tile	10.06	47.11
SLM tile	21.61	55.12

As with area, the NoC power value applies to one domain-socket instance. A preliminary SoC-level sum must account for one instance at each of the four tile positions.

The synthesis results do not cover the custom accelerator tile or the chip-level IO ring. Their areas were estimated separately. The proprietary eFPGA occupies approximately 9 mm^2 according to its LEF boundary. The top level requires 36 external connections, each with a GPIO cell and a bond pad, together with periodically inserted bias cells. These known IO components occupy approximately 0.52 mm^2. This estimate excludes corner cells, spacers, dummy cells, and any structures whose number depends on the final floorplan. It also excludes the surrounding top-level logic.

5.6 Discussion and Limitations

The case study shows that the new technology configuration supports the intended logic-synthesis use case. The existing ESP EDA scripts read the generated TCL database without manual edits, and Genus completed the four unique synthesis targets using the new 28 nm technology. Together with the utility tests described in Section 5.3, these runs show that the configuration mechanism works through synthesis. Metadata used only by physical design or signoff has not yet been tested.

The area results also show why the memory configuration must be considered early. Standard-cell logic forms a small part of the SLM and IO tiles because their SRAMs dominate the mapped area. The choice of available memory macros therefore affects the estimate more strongly than the surrounding logic in these two partitions. The eFPGA is larger than any synthesized partition, but its value is only a LEF footprint and does not include the logic or physical margin needed to integrate the custom tile.

Taken together, the three synthesized tiles, four NoC domain-socket instances, the eFPGA footprint, and the known IO cells remain below the 25 mm^2 SoC budget. This partial sum is not a floorplan or a final die-area estimate. The synthesis reports measure mapped tile and socket area. The eFPGA value comes from its abstract physical view,

and the IO value covers only identified pad-related cells. A complete estimate would also need the custom-tile interface logic, the top-level interconnect and control logic, placement whitespace, macro halos, routing resources, clock distribution, power delivery, and the remaining pad-ring structures. These costs can be determined only after the floorplan has been developed.

The CPU timing result is another open point. The single `cpuerr` path does not prevent the area report from serving its preliminary purpose, but it must be corrected or constrained appropriately before timing closure can be claimed. Its cause has not yet been confirmed. For now, the result identifies an Ibex-specific output path that needs further RTL and constraint review.

The power estimates are preliminary as well. Their fixed activity assumptions do not represent the high-energy-physics workload or the planned eFPGA regression tests. Even with this limitation, leakage deserves attention during physical design: it accounts for roughly one quarter to more than one half of the reported total power, depending on the partition. Genus made extensive use of the available LVT cells during mapping. The balance between timing and leakage should therefore be reviewed before physical design, potentially through tighter control of the allowed threshold-voltage flavors and cell-selection directives.

The completed work stops at logic synthesis. The custom accelerator tile was not synthesized, complete-SoC RTL simulation is not assessed here, and gate-level netlist simulation remains pending. Placement, routing, extracted timing, power-integrity analysis, and signoff verification were also not performed for the new technology. These tasks belong to the subsequent physical-design effort with Fermilab. The contribution of this chapter is therefore the reusable technology-configuration mechanism, the Ibex-related preparation needed by this SoC, and the preliminary implementation data produced for that handoff.

Chapter 6

Conclusions

This thesis examined how the hierarchical ASIC flow of ESP can be made more reliable and easier to reuse as the platform, the design, and the target technology change. The work focused on three related problems: the propagation of SoC configuration choices, the reuse of the implementation stages outside the complete ESP hierarchy, and the organization of technology-dependent inputs. The results strengthen the existing methodology rather than replace it with a new flow.

RQ1: Robustness and configurability. The first research question asked how the flow could be improved across selected releases, configurations, timing constraints, and PVT corners. Preparing ESP release 2026.1.0 exposed errors and outdated assumptions in the public preparation procedure and the private implementation scripts. After correcting these problems, the Python code used for TCL generation was reorganized to reduce repetition and provide a common base for later changes.

The main result was the conditional generation of the tile and standalone NoC timing constraints. The revised generators follow the selected clock strategy and enabled interfaces, including the different clock relations of the dual-external, single-DCO, and multi-DCO cases. The available PVT corners are now declared explicitly and can be selected for each run instead of being inferred from the library filenames. Regression tests covered the three clock strategies, different interface selections, and representative corner sets. The completed cases reached Level 1 without reported timing violations. RQ1 is therefore answered by making configuration choices propagate directly into the generated flow inputs and by validating them for a stated release and setup. The result does not cover the complete configuration space, top-level conditional constraints, or physical implementation.

RQ2: Reuse for standalone designs. The second research question asked to what extent the flow could be separated from the ESP hierarchy. The analysis showed that the main synthesis and physical-design stages can remain common once the required design inputs are supplied separately. These inputs include the RTL sources and their order, the top-level module, timing constraints, floorplan dimensions, and pin organization. They cannot be obtained safely from the flow alone because they describe the intended behavior and physical interface of the design.

This separation was evaluated with a modified look-ahead NoC router. The extracted `simpleflow` carried the design through synthesis, placement, clock-tree synthesis, routing, post-route timing analysis, and parasitic extraction. No setup or hold violations were reported in the selected analysis view. The experiment also showed that the router floorplan was limited by its 7,610 pins rather than by its logic area. The case study reached Level 2 for the pre-filler routed design, but final GDSII, Calibre DRC and LVS, formal equivalence, and gate-level simulation were not completed. RQ2 is therefore answered as a proof that the main stages can be reused for a standalone block, not as a general flow for arbitrary RTL designs.

RQ3: Technology configuration. The third research question asked how the collateral references and metadata of an additional technology could be represented through a common interface. The resulting ESP Tech Config repository uses a fixed YAML template to define the structure of the technology database. A local `tech.yaml` manifest supplies the values and file references for one technology, and a Python generator translates it into the six TCL files already consumed by the flow. The local manifest remains outside version control, while the common structure and generation mechanism can be maintained in the repository.

The interface was applied to the 28 nm technology used in the Fermilab collaboration. Together with the required support for the Ibex processor and its 32-bit memory organization, it allowed the CPU, I/O, shared-local-memory, and NoC targets to be synthesized for preliminary implementation estimates. The CPU result retained one violating output path and was not presented as timing closed. RQ3 is therefore answered by separating the database structure expected by the flow from the values supplied by each technology. The 28 nm configuration was evaluated only through logic synthesis, so the backend and signoff entries of the template remain untested.

Overall, the ESP case shows how separating system configuration, design-specific inputs, common implementation stages, and technology data can improve hierarchical ASIC flow automation for heterogeneous SoC design.

6.1 Future Work

The first priority is to complete and consolidate the current flow. The top-level SDC generator still needs to follow the selected SoC configuration in the same way as the tile and NoC generators. The Python scripts used for physical design can also be reorganized around `TclWriter` and the common functions introduced in this work. Removing dead code, resolving the remaining TODO comments, and using consistent interfaces across the automation would make later changes easier to introduce and test.

Reaching Level 3 requires a closer integration of physical verification. The current verification flow was developed separately and uses different variable names and conventions from the main implementation flow. Its stages should therefore be ported to the current structure, with a common configuration for DRC, LVS, and the other checks required by the selected technology. Once this integration is complete, one representative design can be carried through the full methodology to test the path from its initial configuration to signoff.

Further work is also needed for the ASIC features that are only partly supported. For MSV, the next step is to generate the UPF description automatically, starting from the template already available in the reference `esp_flow_test` repository. For PSO, the main open question is how to add power shut-off without losing modularity as the tiles and power domains change between SoC configurations. The existing DFT support should first be tested on a block-level design. After that, the scan chains and related test logic can be connected at the top level.

Threshold-voltage library selection should also be completed. The standalone router experiment showed that using only one flavor can change the available filler cells and leave gaps that the present finishing sequence cannot fill. Library selection must therefore propagate not only to synthesis and timing analysis, but also to the filler rules used during physical design.

The standalone `simpleflow` must also be prepared before it can be used by the group. The current version shows that the main implementation stages can be reused, but its entry points still contain information written specifically for the router. A usable version should accept the sources, compilation order, top-level module, timing constraints, physical objectives, and pin organization of each custom design. The Vortex GPU is one possible test case, but no next design has yet been selected. The future chiplet support being added to the ESP RTL offers another possible case study. Applying the hierarchical flow to a multi-die system would show which parts of the present block

methodology can be retained and which new physical interfaces and integration steps are required.

For the 28 nm Fermilab project, the remaining task within the Columbia SLD Group is to investigate the violating `cpuerr` path in the Ibex tile and finishing with gate-level netlist simulations. Fermilab will carry out the physical implementation and signoff of the system.

In the longer term, selected stages of the flow could be evaluated with open-source EDA tools. A direct migration of the complete system is unlikely to be simple. The mixed-language RTL limits the choice of frontend tools, an open PDK must provide suitable collateral, and ready-made OpenROAD or OpenLANE-based flows may not support the hierarchy and implementation needs of a complete ESP system. A more practical starting point would be a smaller standalone block or a mixed flow in which open-source tools are introduced only where the design, language, and technology inputs are supported.

6.2 Use of Artificial Intelligence in the Thesis

AI tools were used during both the development work and the writing of this thesis. Their use was permitted for this work and is described here to make their role clear. The main tool was Codex, using the GPT-5.5 and GPT-5.6 Sol models. It operated as an agent with access to the project repository, so its contribution went beyond isolated questions or text completion. The following paragraphs explain what the agent was allowed to do, how its output was checked, and which tasks were completed without it.

For software development, the usual interaction began with a planning phase. The implementation objective was defined by the author, after which possible approaches were discussed with the agent. Once an approach was selected, the agent could inspect the relevant source files, implement the change, and add internal tests. This process was used to understand the repository, investigate failures, and shorten the path from an idea to a working implementation. AI assistance had a substantial role in the Python refactoring and in several bug fixes. It was also used to implement the configurable PVT-corner selection and the ESP Tech Config repository, starting from the requirements and design choices defined during the work.

Access to the repository did not include confidential technology files, implementation reports, or other sensitive material. Repository-level `AGENTS.md` instructions identified the directories and data that the agent was not allowed to inspect. Tasks that required

this information were kept outside the AI-assisted workflow, and the agent was not used to interpret EDA reports. This boundary was one reason why `simpleflow`, which brought together implementation scripts, reports, and outputs, was developed without AI assistance. The conversation records are retained on the Columbia workstation, but they are not included as supplementary material to the thesis.

Passing an agent-generated test was not enough to accept a code change. The agent could create and execute the tests available in its environment, but the resulting changes were reviewed and the relevant Make commands were run again by the author. The simulations, synthesis runs, and physical-design experiments reported in the thesis remained the basis for deciding whether a change worked. AI could therefore assist with implementation and local testing, while the final validation still followed the tools and procedures used for the rest of the project.

During writing, AI was used to prepare initial drafts, revise existing text, correct English, organize the reasoning within sections, prepare tables, place figures, and help with the bibliography. Literature work was based on a small offline collection of papers related to the thesis background, most of which had already been gathered before drafting. The agent could search this collection and insert references while writing. A separate agent pass then checked that the sources existed and supported the surrounding text, and the author performed a final check against the original papers. This three-step process reduced the risk of accepting a plausible but incorrect reference. An AI-generated citation was never treated as evidence on its own.

The experience showed that the quality of AI assistance depended strongly on the task. For programming, it was effective at exploring unfamiliar code, proposing different implementations, finding possible corner cases, and reducing the time between an idea and its first test. Its main weakness was consistency across the repository. When two files needed similar changes, the agent could produce two working but different implementations instead of recognizing that they should share one reusable solution. The timing-constraint generators exposed a more serious limit: successive AI changes often fixed one problem while introducing another. When this did not improve after further iterations, the generated work was removed and the SDC automation was implemented entirely by the author. For writing, AI was useful for English revision and structure, but its first drafts were often too long, vague, or artificial. They required repeated instructions and a final rewrite to match the language and emphasis of the thesis.

These differences make it difficult to describe AI as either a replacement for research work or only a writing aid. In this thesis, it was most useful when combined with

a clear objective, controlled access to the project, and results that could be checked independently. It accelerated implementation and helped examine alternatives, but it did not replace the decisions about what to build, the handling of confidential data, the execution of the final experiments, or responsibility for the text and conclusions. For future theses, a useful disclosure should therefore explain the tasks assigned to AI, the information available to it, the checks applied to its output, and the cases in which that output was rejected. This gives a clearer account of the student's work than a simple statement that AI was or was not used.

References

[1] Gordon E. Moore. “Cramming More Components onto Integrated Circuits”. In: *Electronics* 38.8 (Apr. 1965), pp. 114–117.

[2] Robert H. Dennard et al. “Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[3] Mark T. Bohr and Ian A. Young. “CMOS Scaling Trends and Beyond”. In: *IEEE Micro* 37.6 (Nov. 2017), pp. 20–29. DOI: 10.1109/MM.2017.4241347.

[4] John L. Hennessy and David A. Patterson. “A New Golden Age for Computer Architecture”. In: *Communications of the ACM* 62.2 (Feb. 2019), pp. 48–60. DOI: 10.1145/3282307.

[5] Greg Yeric et al. “The Past, Present, and Future of Design-Technology Co-Optimization”. In: *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference*. IEEE, Sept. 2013, pp. 1–8. DOI: 10.1109/CICC.2013.6658476.

[6] Karl Rupp. *50 Years of Microprocessor Trend Data*. Licensed under the Creative Commons Attribution 4.0 International License. Feb. 22, 2022. URL: https://github.com/karlrupp/microprocessor-trend-data (visited on 09/02/2026).

[7] Hadi Esmaeilzadeh et al. “Dark Silicon and the End of Multicore Scaling”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.

[8] International Technology Roadmap for Semiconductors. *2001 Edition: System Drivers*. Roadmap. Semiconductor Industry Association, 2001. URL: https://www.semiconductors.org/wp-content/uploads/2018/08/2001SysDrivers.pdf (visited on 09/02/2026).

[9] William J. Dally, Yatish Turakhia, and Song Han. “Domain-Specific Hardware Accelerators”. In: *Communications of the ACM* 63.7 (July 2020), pp. 48–57. DOI: 10.1145/3361682.

[10] Luca P. Carloni. “The Case for Embedded Scalable Platforms”. In: *Proceedings of the 53rd Annual Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–6. DOI: 10.1145/2897937.2905018.

[11] Paolo Mantovani et al. “Agile SoC Development with Open ESP”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–9. DOI: 10.1145/3400302.3415753.

[12] Neil H. E. Weste and David Money Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4th ed. Boston, MA, USA: Addison-Wesley, 2011. ISBN: 978-0-321-54774-3.

[13] Maico Cassel dos Santos. “Scalable Chip Design for Tile-Based Heterogeneous Architectures”. PhD thesis. Columbia University, 2025. DOI: 10.7916/mbjr-ts62.

[14] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. New York, NY, USA: McGraw-Hill, 1994. ISBN: 978-0-07-016333-1.

[15] Cadence Design Systems. *Are You Wondering What Logic Synthesis Is and How to Deal with Its Complex Process? Relax!* Oct. 19, 2021. URL: https://community.cadence.com/cadence_blogs_8/b/cadence-support/posts/are-you-wondering-what-logic-synthesis-is-and-how-to-deal-with-its-complex-process-relax (visited on 08/13/2026).

[16] Cadence Design Systems. *Timing and Mode Setup*. Cadence Design Systems. Nov. 1, 2010. URL: https://community.cadence.com/cfs-file/___key/telligent-evolution-components-attachments/00-29-01-00-01-33-26-59/Timing-and-Mode-Setup.v1.pdf (visited on 09/02/2026).

[17] Andrew B. Kahng et al. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Dordrecht, The Netherlands: Springer, 2011. ISBN: 978-90-481-9590-9. DOI: 10.1007/978-90-481-9591-6.

[18] J. Bhasker and Rakesh Chadha. *Static Timing Analysis for Nanometer Designs: A Practical Approach*. New York, NY, USA: Springer, 2009. ISBN: 978-0-387-93819-6. DOI: 10.1007/978-0-387-93820-2.

[19] Ahmed Elzeftawi and Ken Tseng. *Addressing Process Variation and Reducing Timing Pessimism at 16nm and Below*. White paper. Cadence Design Systems, 2019. URL: https://login.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/custom-ic-analog-rf-design/library-characterization-addressing-process-variation-wp.pdf (visited on 09/02/2026).

[20] Cadence Design Systems. *Innovus Implementation System*. Cadence Design Systems. Aug. 2019. URL: https://login.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/innovus-implementation-system-ds.pdf (visited on 09/02/2026).

[21] Cadence Design Systems. *Virtuoso Digital Implementation*. Cadence Design Systems. July 2013. URL: https://login.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/custom-ic-analog-rf-design/virtuoso-digital-implementation-ds.pdf (visited on 09/02/2026).

[22] Pawan Gandhi et al. *Hierarchical Timing Analysis: Pros, Cons, and a New Approach*. White paper. Cadence Design Systems, May 2020. URL: https://login.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/hierarchical-timing-analysis-wp.pdf (visited on 09/02/2026).

[23] SEMI. *Specification for OASIS—Open Artwork System Interchange Standard*. Standard SEMI P39-0416. SEMI, Apr. 2016. URL: https://store-us.semi.org/products/p03900-semi-p39-specification-for-oasis%C2%AE-open-artwork-system-interchange-standard (visited on 09/02/2026).

[24] Maico Cassel dos Santos et al. "A Scalable Methodology for Agile Chip Development with Open-Source Hardware Components". In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1–9. DOI: 10.1145/3508352.3561102.

[25] Young Jin Yoon et al. "Virtual Channels and Multiple Physical Networks: Two Alternatives to Improve NoC Performance". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.12 (Dec. 2013), pp. 1906–1919. DOI: 10.1109/TCAD.2013.2276399.

[26] Davide Giri et al. "ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1049–1054. DOI: 10.23919/DATE48585.2020.9116317.

[27] Tianyu Jia et al. "A 12nm Agile-Designed SoC for Swarm-Based Perception with Heterogeneous IP Blocks, a Reconfigurable Memory Hierarchy, and an 800MHz Multi-Plane NoC". In: *ESSCIRC 2022—IEEE 48th European Solid State Circuits Conference*. IEEE, 2022, pp. 269–272. DOI: 10.1109/ESSCIRC55480.2022.9911456.

[28] Maico Cassel dos Santos et al. "A 12nm Linux-SMP-Capable RISC-V SoC with 14 Accelerator Types, Distributed Hardware Power Management and Flexible NoC-Based Data Orchestration". In: *2024 IEEE International Solid-State Circuits*

Conference (ISSCC). Vol. 67. IEEE, 2024, pp. 262–264. DOI: 10.1109/ISSCC49657.2024.10454572.

[29] Python Software Foundation. *__main__—Top-Level Code Environment*. Version Python 3.14.7 documentation. Aug. 5, 2026. URL: https://docs.python.org/3.14/library/__main__.html (visited on 09/02/2026).

[30] Guido van Rossum and Alyssa Coghlan. *PEP 343—The "with" Statement*. 2005. URL: https://peps.python.org/pep-0343/ (visited on 08/18/2026).

[31] Eric V. Smith. *PEP 498—Literal String Interpolation*. 2015. URL: https://peps.python.org/pep-0498/ (visited on 08/18/2026).

[32] Guido van Rossum, Barry Warsaw, and Alyssa Coghlan. *PEP 8—Style Guide for Python Code*. 2001. URL: https://peps.python.org/pep-0008/ (visited on 08/18/2026).

[33] Columbia University System-Level Design Group. *Embedded Scalable Platform Repository, Release 2026.1.0*. Version 2026.1.0. Commit `a45f2bb`. Mar. 25, 2026. URL: https://github.com/sld-columbia/esp/releases/tag/2026.1.0 (visited on 09/02/2026).