الجمهورية الجزائرية الديمقراطية الشعبية
RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
وزارة التعليم العالي والبحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
École Nationale Polytechnique

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

UNIVERSITÄT ZU LÜBECK

Département d'Électronique
En cotutelle avec l'Universität zu Lübeck
End-of-study project dissertation
For the attainment of the State Engineer Degree in Electronics

Modelling of Complex Hardware Component Models in SystemC/TLM

Prepared by :
OUAREZKI Chaimaa
REMMA Hana

Under the supervision of :
Dr. Mohamed Oussaid TAGHI (ENP, Algiers)
Dr.-Ing. Rolf MEYER (Universität zu Lübeck)

Presented and defended publicly on : 22/06/2026

Examination Board :

President :	Pr. Hicham BOUSBIA-SALAH	ENP, Alger
Supervisor :	Dr.-Ing. Rolf MEYER	Universität zu Lübeck, Germany
Co-Supervisor :	Ms. Celine THERMANN	Universität zu Lübeck, Germany
Co-Supervisor :	Dr. Mohamed Oussaid TAGHI	ENP, Alger
Examiner :	Dr. Nesrine BOUADJENEK	ENP, Alger

ENP 2026

Électronique / ENP, 10 Rue des Frères OUDEK, El Harrach 16200 Alger Algérie.

الجمهورية الجزائرية الديمقراطية الشعبية
RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
وزارة التعليم العالي والبحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
École Nationale Polytechnique

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

UNIVERSITÄT ZU LÜBECK

Département d'Électronique
En cotutelle avec l'Universität zu Lübeck
End-of-study project dissertation
For the attainment of the State Engineer Degree in Electronics

Modelling of Complex Hardware Component Models in SystemC/TLM

Prepared by :
OUAREZKI Chaimaa
REMMA Hana

Under the supervision of :
Dr. Mohamed Oussaid TAGHI (ENP, Algiers)
Dr.-Ing. Rolf MEYER (Universität zu Lübeck)

Presented and defended publicly on : 22/06/2026

Examination Board :

President :	Pr. Hicham BOUSBIA-SALAH	ENP, Alger
Supervisor :	Dr.-Ing. Rolf MEYER	Universität zu Lübeck, Germany
Co-Supervisor :	Ms. Celine THERMANN	Universität zu Lübeck, Germany
Co-Supervisor :	Dr. Mohamed Oussaid TAGHI	ENP, Alger
Examiner :	Dr. Nesrine BOUADJENEK	ENP, Alger

ENP 2026

Électronique / ENP, 10 Rue des Frères OUDEK, El Harrach 16200 Alger Algérie.

الجمهورية الجزائرية الديمقراطية الشعبية
RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
وزارة التعليم العالي والبحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
École Nationale Polytechnique

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

UNIVERSITÄT ZU LÜBECK

Département d'Électronique
En cotutelle avec l'Universität zu Lübeck
Mémoire de fin détude
Pour l'obtention du diplôme d'ingénieur d'État en électronique

Modélisation des composants complexes en SystemC/TLM

Réalisé par :
OUAREZKI Chaimaa
REMMA Hana

Sous la direction de :
Dr. Mohamed Oussaid TAGHI (ENP, Algiers)
Dr.-Ing. Rolf MEYER (Universität zu Lübeck)

Présenté et soutenu publiquement le : 22/06/2026

Composition du Jury :

Président :	Pr. Hicham BOUSBIA-SALAH	ENP, Alger
Encadrant :	Dr.-Ing. Rolf MEYER	Universität zu Lübeck, Germany
Co-encadrant :	Ms. Celine THERMANN	Universität zu Lübeck, Germany
Co-encadrant :	Dr. Mohamed Oussaid TAGHI	ENP, Alger
Examinatrice :	Dr. Nesrine BOUADJENEK	ENP, Alger

ENP 2026

Électronique / ENP, 10 Rue des Frères OUDEK, El Harrach 16200 Alger Algérie.

Dedication

﴿الْحَمْدُ لِلَّهِ الَّذِي هَدَانَا لِهَٰذَا وَمَا كُنَّا لِنَهْتَدِيَ لَوْلَا أَنْ هَدَانَا اللَّهُ﴾

To my dear father and mother: *The fountains of endless giving and unwavering support.*

To my brothers: *My finest pillars, who have always protected me, shared their knowledge, and guided me with their experience.*

To my honorable professors: *Who illuminated my path with their knowledge.*

To my friends, and to everyone *I have learned from, learned with, or who supported me throughout my academic journey…*

Chaimâa

Dedication

To my beloved father (PAPA),
the person who has profoundly shaped my life and the woman I have become today. You have always been my role model, my source of strength, and my greatest support. From a very young age, you taught me to believe in myself, to persevere, and never to give up in the face of difficulties. Your values, your wisdom, and your love continue to guide me every day. I am immensely proud to be your daughter, and I will always be grateful for everything you have given me. You will forever remain my greatest inspiration and the person I admire most.

With all my love, gratitude, and respect, I dedicate this work to everyone who supported, encouraged, and believed in me throughout this journey.

HR

Acknowledgments

First and foremost, we would like to express our deepest and most sincere gratitude to ***Ms. Celine Thermann****. Her support, guidance, patience, and dedication throughout this work went far beyond the role of a supervisor. She was always available to help, encourage, and guide us through every challenge we encountered. Her trust, kindness, and unwavering commitment were invaluable, and this work would not have been possible without her continuous support. We are profoundly grateful for everything she has done for us throughout this journey.*

We would also like to sincerely thank ***Dr. Rolf Meyer*** *for giving us the opportunity to be part of his team and for sharing his knowledge, expertise, and valuable guidance throughout this project.*

Our heartfelt thanks go to ***Mr. Mohamed Taghi*** *for his dedication to the Electronics Department, his continuous support throughout our years of study, and his valuable guidance during this project.*

We would like to express our sincere gratitude to ***Prof. Hicham Bousbia-Salah*** *for accepting to serve as President of the Jury, and to* ***Ms. Nesrine Bouadjenek*** *for accepting to examine and evaluate this work.*

Finally, we would like to thank everyone who contributed, directly or indirectly, to the completion of this project and supported us throughout this journey.

ملخص

يتطلّب بناء النماذج الأولية الافتراضية بصيغة SystemC/TLM-2.0 نسخ خرائط السجلّات يدويًا من أدلّة المرجع التقنية، وهو عائق معرّض للأخطاء. يُؤتمِت هذا العمل تلك المهمّة عبر خطّ معالجة يحوّل الوثائق غير المهيكَلة إلى نماذج SystemC/TLM-2.0، بالجمع بين التحليل بنموذج لغوي كبير (LLaMA 3.3 70B) والتطبيع المُتحقَّق منه بالمُصرِّف وفق معيار SystemRDL، مقلّصًا الجهد من ساعات إلى دقائق. وعلى ثلاثة أطراف محيطية من متحكّم ESP32-P4، بلغت الدقّة الهيكلية 94.1--99.6%، في حين تطلّب التوافق الكامل مع البرامج الثابتة تحسينًا سلوكيًا يدويًا.

الكلمات المفتاحية: النمذجة الأولية الافتراضية، النماذج اللغوية الكبيرة، SystemRDL، SystemC/TLM-2.0، ESP32-P4.

Résumé

La construction de prototypes virtuels SystemC/TLM-2.0 exige la transcription manuelle des cartes de registres depuis les manuels de référence techniques, un goulot d'étranglement sujet aux erreurs. Ce mémoire l'automatise par un pipeline qui transforme une documentation non structurée en modèles SystemC/TLM-2.0, via l'analyse par un grand modèle de langage (LLaMA 3.3 70B) et une normalisation SystemRDL validée par compilateur, réduisant l'effort de plusieurs heures à quelques minutes. Sur trois périphériques de l'ESP32-P4, il atteint 94,1–99,6 % de précision structurelle ; la compatibilité complète avec le micrologiciel a nécessité un raffinement comportemental manuel.

Mots clés : prototypage virtuel, grands modèles de langage, SystemRDL, SystemC/TLM-2.0, ESP32-P4.

Abstract

Building SystemC/TLM-2.0 virtual prototypes requires manually transcribing register maps from Technical Reference Manuals, an error-prone bottleneck. This thesis automates it with a pipeline that turns unstructured documentation into SystemC/TLM-2.0 models via Large Language Model parsing (LLaMA 3.3 70B) and compiler-validated SystemRDL, cutting effort from hours to minutes. On three ESP32-P4 peripherals it reached 94.1–99.6% structural accuracy; full firmware compatibility additionally required manual behavioral refinement.
Your English abstract text goes here.

Keywords: virtual prototyping, Large Language Models, SystemRDL, SystemC/TLM-2.0, ESP32-P4.

Contents

List of Figures

List of Tables

List of Acronyms

General Introduction 13

1 Research Framework 15
1.1 Motivation and Problem Statement 15
1.2 Key Technical Challenges and Research Objectives 16
1.3 Related Work 17
1.4 Conclusion 18

2 Technical Background and Research Context 20
2.1 Virtual Platforms and the ESP32-P4 20
2.2 SystemC and Transaction-Level Modelling 22
2.3 Register Modelling and SystemRDL 23
2.4 Conclusion 24

3 Automated Extraction and SystemC/TLM Generation 25
3.1 Automation Pipeline Overview 25
3.2 TRM Document Preprocessing 27
3.3 LLM-Based Register Extraction 29
3.4 Model Selection: LLaMA 3.3 70B on Groq 33
3.5 Prompt Engineering 34
3.6 SystemRDL Generation and Validation 36
3.7 Automated SystemC/TLM Peripheral Generation 38
3.8 Experimental Results and Discussion 42
3.9 Conclusion 45

4 Implementation Evaluation and Functional Verification 46
4.1 Experimental Environment and Development Infrastructure 47
4.2 Platform Preparation: Manual Modeling of CLIC and ISP 48
4.3 Phase 1: Structural Evaluation of Auto-Generated Peripherals 51
4.4 Phase 2: Behavioral Evaluation of Manually Modeled Peripherals 53
4.5 Comparison of Phase 1 and Phase 2 Approaches 57
4.6 Comparison of the Two Approaches 59
4.7 Productivity and Scalability 59
4.8 Generalisation and Reusability 61
4.9 General Discussion 61
4.10 Conclusion 63

5 Synthesis and Future Work 64
5.1 Thesis Summary and Key Achievements 64
5.2 Synthesis of Findings across Evaluated Peripherals 65
5.3 Limitations and Open Challenges 66
5.4 Directions for Future Work 67
5.5 Conclusion 68

Bibliography 69

List of Figures

Figure 1.1 Traditional sequential design flow versus shift-left verification with a virtual prototype. 16
Figure 2.1 Modelling abstraction levels and where virtual platforms sit. 21
Figure 2.2 Architecture of the RISC-V virtual platform targeting the ESP32-P4. 22
Figure 2.3 SystemC structural hierarchy: a CPU and a memory module connected by `sc_signal`channels inside a top-level module. 23
Figure 2.4 Signal-level (RTL) versus transaction-level (TLM-2.0) communication. 23
Figure 2.5 Register organisation and the corresponding SystemRDL description for a RISC-V based peripheral. 24
Figure 3.1 Block diagram of the six-stage pipeline from TRM PDF to SystemC/TLM. 26
Figure 3.2 PDF preprocessing pipeline for TRM register extraction. 27
Figure 3.5 Single-call extraction (left) versus chunked extraction (right). 30
Figure 3.8 Effect of the temperature parameter on deterministic LLM inference. 32
Figure 3.13 Per-register SystemRDL generation with an isolated retry path. 36
Figure 3.14 Compiler validation and LLM-based repair loop for generated SystemRDL. 37
Figure 3.16 Four-layer architecture of a generated SystemC/TLM peripheral module. 39
Figure 4.1 Evaluation flow: Structure and Behavior models. 46
Figure 4.2 Manual modeling reveals the automation boundary. 48
Figure 4.3 Interrupt Matrix (CLIC) functional flow. 49
Figure 4.5 ISP processing pipeline and VDMA integration 50
Figure 4.7 Bipartite ESL Modeling Flow: Automated Model of Structure (MoS) Generation via LLM and SystemRDL combined with Manual Model of Behavior (MoB) for complete peripheral validation. 52
Figure 4.8 Phase 2 behavioural evaluation flow: hand-written UART, I2C, and LEDC TLM models integrated into the RISC-V virtual platform and validated using the same bare-metal firmware as Phase 1. 54
Figure 4.9 Comparison of Guest Firmware Interactivity: Static Register Map (Phase 1) vs. Reactive Behavioral Model (Phase 2) with Write Callbacks to Prevent Driver Deadlocks. 56
Figure 4.10 Parametric aperture masking and bitwise offset truncation for mapping local storage-array indices. 60

List of Tables

Table 3.10 Comparison of the five evaluated LLM models for register extraction. 34
Table 3.15 SystemRDL compiler success rates before and after automatic repair. 37
Table 3.19 Comparison of the generated UART, I2C, and LEDC modules. 42
Table 3.20 Per-peripheral extraction results using Groq LLaMA 3.3 70B. 43
Table 3.21 Detailed per-metric extraction results across the three peripherals. 43
Table 3.22 Taxonomy of the extraction errors observed across the peripherals. 44
Table 4.4 CLIC validation: testbench scenarios and outcomes. 49
Table 4.6 ISP validation on synthetic gradient images (160 × 120). 51

List of Acronyms

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
AST	Abstract Syntax Tree
AXI	Advanced eXtensible Interface
CLIC	Core Local Interrupt Controller
EDA	Electronic Design Automation
ELF	Executable and Linkable Format
ESL	Electronic System Level
FIFO	First-In First-Out
GDB	GNU Debugger
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
I2C	Inter-Integrated Circuit
IoT	Internet of Things
IP	Intellectual Property
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISP	Image Signal Processor
ISS	Instruction Set Simulator
JSON	JavaScript Object Notation
LEDC	LED PWM Controller
LLM	Large Language Model
LPU	Language Processing Unit
MoB	Model of Behavior
MoS	Model of Structure
OTG	On-The-Go
PWM	Pulse Width Modulation
RAG	Retrieval-Augmented Generation
RAM	Random Access Memory
RISC-V	Reduced Instruction Set Computer V
RTL	Register-Transfer Level
SoC	System on Chip
SSoT	Single Source of Truth
SystemC	SystemC Language Reference Manual
SystemRDL	System Register Description Language
TLM	Transaction Level Modeling

TRM	Technical Reference Manual
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
UVM	Universal Verification Methodology
VP	Virtual Platform

General Introduction

Modern embedded systems are increasingly built around complex **Systems-on-Chip (SoCs)** that integrate multiple cores, memories, and peripherals on a single die, such as the dual-core RISC-V Espressif ESP32-P4. A recurring difficulty in such designs is that the software for a new chip can normally only be tested once the physical silicon is available, and that hardware arrives late in the project, so any defect discovered at that stage is costly to fix. To break this dependency, development teams rely on **shift-left verification**, in which firmware is developed against a **virtual prototype** — an executable software model of the chip, most often built with **SystemC** and **TLM-2.0** — long before silicon exists.

The cost of this approach lies in building the models themselves. A virtual platform must describe hundreds of configuration registers, traditionally transcribed by hand: an engineer reads the vendor Technical Reference Manual (TRM), interprets each register layout, and manually translates it into SystemC/TLM C++ code. This process is tedious, repetitive, and error-prone, and has become one of the main bottlenecks in **Electronic System-Level (ESL) design**.

This thesis addresses that bottleneck with an **automated pipeline** that synthesises SystemC/TLM-2.0 peripheral models directly from unstructured technical documentation. A **Large Language Model (LLM)** acts as a semantic parser that extracts register descriptions from TRM PDFs and normalises them into **SystemRDL**, a standardised register-description language that imposes a strict, machine-verifiable structure. From this validated representation, a Python generator automatically produces integration-ready SystemC/TLM models, reducing the modelling effort from **engineering hours to computational minutes**.

The objectives of the work are to automate the extraction of register descriptions from vendor documentation, to formalise them in a compiler-validated SystemRDL representation, to generate integration-ready SystemC/TLM-2.0 models, and to evaluate the approach by integrating the generated peripherals into a RISC-V virtual platform of the ESP32-P4 and measuring their functional accuracy. The methodology is validated on three peripherals with differing difficulty profiles — a **UART**, an **I2C controller**, and a **LED PWM controller (LEDC)** — through a two-phase evaluation covering structural integrity and behavioural fidelity.

Thesis Organization

The body of this thesis is organised into five chapters that build on one another, progressing from the necessary background, through the proposed pipeline, to its experimental evaluation. It begins with a general introduction that outlines the field and its main

motivations.

- **Chapter 1** motivates the research domain, outlines the key architectural challenges, and summarises the thesis objectives and core contributions.
- **Chapter 2** details the technical background required for this work, focusing on the Espressif ESP32-P4 microcontroller architecture, transaction-level modelling with SystemC and TLM-2.0, and the SystemRDL register-description standard.
- **Chapter 3** describes the automated tool pipeline in depth, explaining the document semantic-extraction strategies and the implementation logic behind the generator.
- **Chapter 4** evaluates the automated framework empirically, presenting the simulation results, analysing the generation-performance metrics, and detailing how the platform-integration bugs were resolved.
- **Chapter 5** synthesises and future work,discusses the current technical limitations, and maps out directions for future work.

Finally, a general conclusion closes the thesis, drawing together its main outcomes and overall contribution.

1

Research Framework

This chapter frames the research — its context, motivation, and objectives. As embedded systems integrate ever more heterogeneous components, traditional sequential design flows stall on late software verification. This work addresses that gap with an automated, end-to-end pipeline that uses Large Language Models (LLMs) and the SystemRDL standard to synthesise accurate SystemC/TLM-2.0 virtual prototypes directly from unstructured technical documentation, evaluated on the Espressif ESP32-P4 microcontroller. Section 1.1 sets out the problem, Section 1.2 the challenges and objectives, and Section 1.3 the related work.

1.1 Motivation and Problem Statement

Modern products are built around Systems-on-Chip (SoCs) such as Espressif's ESP32-P4, which integrate multiple cores, fast memory, and diverse communication interfaces on a single die. Their software, however, can only run on the real device once silicon is available — and that silicon arrives late, after design, fabrication, and shipping, a cycle of weeks or months. A bug found at that stage is costly to fix and threatens the entire schedule [1].

To break this dependency, teams adopt *shift-left* verification: firmware is developed and tested against a *virtual prototype* long before silicon exists (Figure 1.1). A virtual prototype is a software model that runs the real target binaries on an ordinary computer, so that bare-metal firmware, drivers, and applications can be brought up early [2]. Such models are most often built with SystemC and TLM-2.0.

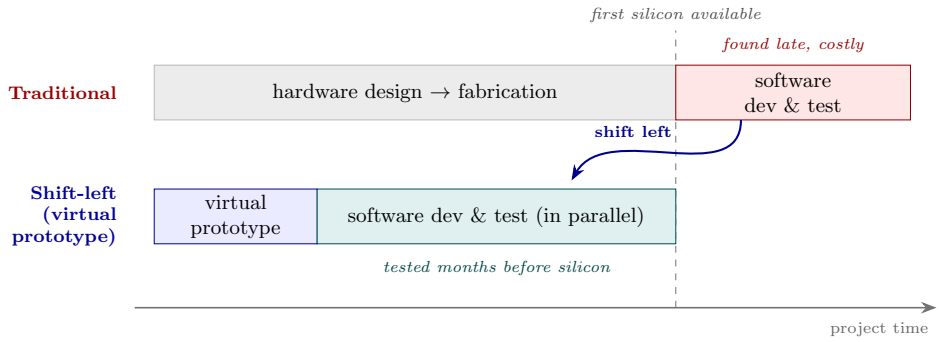


Figure 1.1: Traditional sequential design flow versus shift-left verification with a virtual prototype.

The bottleneck is building the prototype itself. Conventionally, an engineer reads the vendor Technical Reference Manual (TRM), interprets each register configuration, and transcribes it by hand into SystemC/TLM-2.0 C++. Because the layout is fixed at compile time, every specification change restarts a cycle of manual edits, recompilation, and re-linking, scaling in hours or days per peripheral and delaying co-verification. This thesis replaces that manual step with an automated pipeline, described next.

1.2 Key Technical Challenges and Research Objectives

Automating this process with current parsing techniques and LLMs raises two distinct challenges.

Representation bottleneck. Raw text and tables from semantic extraction cannot be turned reliably into syntactically correct SystemC. SystemC and TLM-2.0 are the de-facto standard for transaction-level prototyping [2], but — like the SystemRDL intermediate format — they remain niche technologies on which LLMs have seen little training data. Asked to emit complex template code directly from unstructured text, an LLM frequently introduces syntax errors, incorrect memory offsets, or omits reserved register locations — deviations that hardware modelling cannot tolerate [3].

Platform integration. Generated components often fail to communicate with the rest of the system, for two reasons: an address-decoding mismatch between absolute bus addresses and the local, register-relative offsets used inside a peripheral, and an interface incompatibility when peripheral interrupt lines must be connected to the core's interrupt controller [4].

To address these challenges, this thesis proposes an automated framework that converts technical documentation into functionally accurate virtual prototypes without manual data entry, targeting the Espressif ESP32-P4 [5]. An LLM acts as a semantic parser that translates PDF datasheets into SystemRDL, a standardised register-description language that imposes a strict structural boundary on the extracted data [6, 7]. A custom Python generator then synthesises executable SystemC/TLM-2.0 models from this representation [8, 10], enforcing hardware constraints through statically computed read and write masks,

and integrates them into an existing RISC-V virtual platform that executes bare-metal C binaries [11] — enabling the analysis of realistic integration problems rather than isolated components.

Research Objectives

This work pursues four objectives:

1. Automate the extraction of register descriptions from vendor documentation through an LLM-based pipeline.
2. Formalise the extracted information in a compiler-validated SystemRDL intermediate representation.
3. Generate integration-ready SystemC/TLM-2.0 peripheral models from that representation.
4. Evaluate the approach by integrating the generated peripherals into a RISC-V virtual platform of the ESP32-P4 and measuring their functional accuracy.

Contributions

The main technical contributions are:

- **LLM-driven semantic parsing pipeline** that extracts register descriptions from unstructured datasheets, removing manual layout entry.
- **Strict intermediate-representation normalisation** that bridges probabilistic LLM outputs into deterministic, compiler-validated SystemRDL specifications.
- **Automated target-model synthesis** that maps SystemRDL directly into standard-compliant SystemC/TLM-2.0 peripherals, cutting modelling overhead from engineering hours to computational minutes.
- **Full integration evaluation** that embeds the synthesised components into a running RISC-V virtual platform executing real target binaries.

1.3 Related Work

LLMs for hardware design. Most work applying LLMs to hardware targets the Register-Transfer Level (RTL). RTLCoder [12], ChatChisel [13], and ReChisel [14] show that iterative refinement with compiler feedback yields high syntactic correctness, while VerilogEval [15] and GenBen [16] benchmark RTL generation and reveal that even strong models struggle with functional correctness on complex designs. RTL's cycle-accurate simulation, however, is too slow for the early firmware co-design that motivates transaction-level virtual platforms.

LLMs for transaction-level models. Applying LLMs to high-level reference models is less explored. ChatModel [17] targets SystemC/C++ reference-model automation, and iDSE [18] explores LLMs in high-level synthesis. Abdollahi et al. [19] identify SystemC modelling as a persistent bottleneck, where general-purpose models struggle with transaction-level semantics, concurrency, and bit-width management — consistent with our error tax-

onomy, in which hardware-specific issues such as parametric offsets and cross-referenced reset values do not appear in generic benchmarks.

TLM-2.0 and platform infrastructure. Transaction-Level Modelling enables simulation fast enough to boot complete software stacks. Grötker et al. [2] established the core SystemC concepts, and IEEE 1666-2023 [20] formalised the delta-cycle timing and event-driven concurrency that any generated SystemC must respect. Braun et al. [21] demonstrated rapid prototyping for shift-left verification, while the RISC-V Virtual Platform [11] and Machineware [22] provide the open-source SystemC/TLM infrastructure into which generated peripherals are integrated — the level at which address decoding, synchronisation, and interrupt routing actually surface.

Register-specification standards. Accellera's SystemRDL 2.0 [6] defines a machine-readable syntax for register maps, bit ranges, reset values, and access permissions. Unlike natural-language datasheets, it supports compiler validation and deterministic code generation. Our methodology enforces SystemRDL as a validation checkpoint, separating semantic extraction (the LLM) from deterministic, template-based generation, which limits the propagation of hallucinations into executable code.

Repair and validation. Xia et al. [23] show that conversation-based repair with compiler feedback resolves bugs efficiently, a paradigm that informs our compiler-driven validation loop (Section 3.6), where SystemRDL compilation errors trigger automatic repair. ChatTest [24] adds coverage-enhanced testbench generation, and RefEvo [25] identifies the *coupled validation failure* problem, where design and testbench share hallucinations that falsely verify a flawed model. Our Phase 2 evaluation (Section 4.4) confirms that structural correctness alone is insufficient: behavioural side-effects require manual modelling to prevent firmware deadlocks.

Toward adaptive pipelines. Recent work moves toward multi-agent frameworks. RefEvo [25] introduces dynamic planning, co-evolutionary verification, and spec anchoring, and TDAG [26] argues that task complexity should drive the workflow. These suggest future extensions to our fixed six-stage pipeline: complexity-aware chunk sizing, co-evolutionary detection of behavioural errors, and Retrieval-Augmented Generation to resolve cross-referenced constants [27].

Positioning. This thesis differs from prior work in that it (1) provides complete end-to-end automation from unstructured TRM PDFs to SystemC/TLM models, (2) enforces SystemRDL as a compiler-validated intermediate representation, (3) enables practical shift-left verification on a commercial SoC, and (4) contributes a systematic error taxonomy (Section 3.8).

1.4 Conclusion

This chapter framed the research. It established why shift-left verification with SystemC/TLM-2.0 virtual prototypes matters and identified the manual modelling of peripheral registers as the central bottleneck. It then defined the two challenges this work tackles — the representation gap between unstructured documentation and correct SystemC,

and the integration of generated components into a complete platform — and set out the objectives and contributions of an automated, SystemRDL-centred pipeline, positioning it against related work in LLM-assisted hardware design, transaction-level modelling, and register-specification standards. The next chapter presents the pipeline itself, detailing each stage from PDF preprocessing to SystemC/TLM generation.

2

Technical Background and Research Context

This chapter establishes the technical foundation on which the rest of the thesis builds. The goal of the work is to model hardware peripherals as functionally accurate virtual prototypes, so three ingredients must first be introduced: the target platform, the modelling language, and the register-description formalism that links them.

Section 2.1 introduces virtual platforms—what they are, the abstraction levels at which hardware can be modelled, and where transaction-level prototypes sit among them—and presents the Espressif ESP32-P4 microcontroller used as the case study. Section 3.7 describes SystemC and used as the case study. Section 2.2 describes SystemC and Transaction-Level Modelling (TLM-2.0), the framework used to build the executable models, and explains why it offers the speed needed to run real software. Finally, Section 2.3 presents SystemRDL, the standardised register-description language that captures a peripheral's register map in a machine-readable form and serves as the bridge between the documentation and the generated SystemC/TLM code.

2.1 Virtual Platforms and the ESP32-P4

A central concept of this thesis is the *virtual platform* (or virtual prototype): an executable software model of a complete chip that runs the real target binaries on an ordinary host computer. Virtual platforms exist because hardware and software can no longer be developed strictly one after the other; modelling the hardware in software lets firmware and drivers be written, executed, and debugged before any silicon is available [21].

A model never reproduces hardware in full detail. Instead, the designer chooses a *level of abstraction* that trades timing accuracy for simulation speed [28]. Figure 2.1 summarises the common levels. At the register-transfer level (RTL), every signal is evaluated on each clock edge: this is accurate enough for hardware synthesis but far too slow to boot an operating system. At the transaction level (TLM), the model abstracts away individual pins and clock edges and is fast enough to run complete software stacks. Virtual platforms therefore sit at the transaction level; instruction-set simulators and tools such as QEMU occupy the more abstract, purely functional end of the spectrum, while RTL simulation lies at the detailed end and is reserved for hardware verification.

Beyond early software development, virtual platforms serve several purposes. They

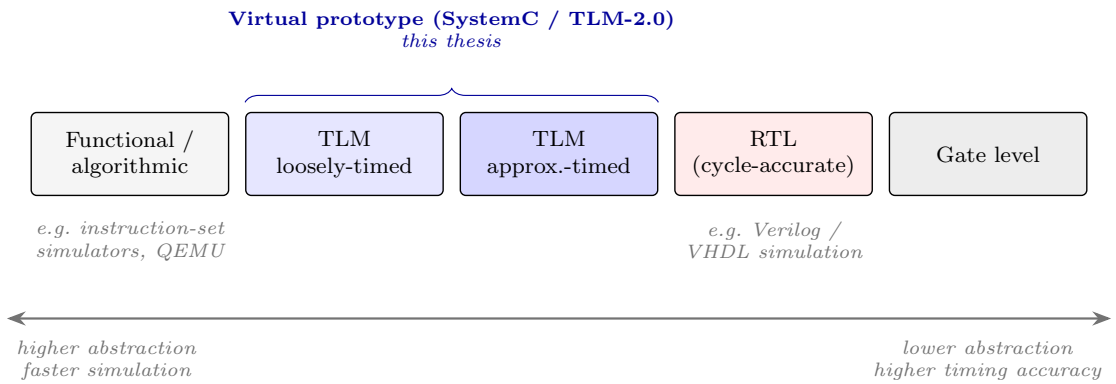


Figure 2.1: Modelling abstraction levels and where virtual platforms sit.

give full visibility into internal registers and bus traffic, which greatly simplifies debugging; they allow architectural exploration, since memory maps and peripherals can be changed without re-spinning hardware; and they can be integrated into continuous-integration pipelines so that firmware is tested automatically on every change [21].

The platform modelled in this work targets the Espressif ESP32-P4, a representative modern system-on-chip for embedded and IoT applications. It was chosen because it combines a dual-core 32-bit RISC-V processor with a rich set of standard peripherals on a single die, and because its RISC-V instruction set is open, which makes an accurate simulator feasible without proprietary models. Figure 2.2 shows the architecture of the virtual platform: the RISC-V cores, the L2 memory, and the peripheral subsystem are connected through an AMBA AXI/AHB-Lite interconnect that carries every memory access. Software controls each peripheral through *memory-mapped registers*: reading or writing a fixed address on the bus configures the device, starts a transfer, or reads its status. The peripherals modelled and evaluated in this thesis—a UART, an I2C controller, and an LED PWM controller (LEDC)—are all driven in exactly this way, which is why faithful register models are the foundation of the whole platform.

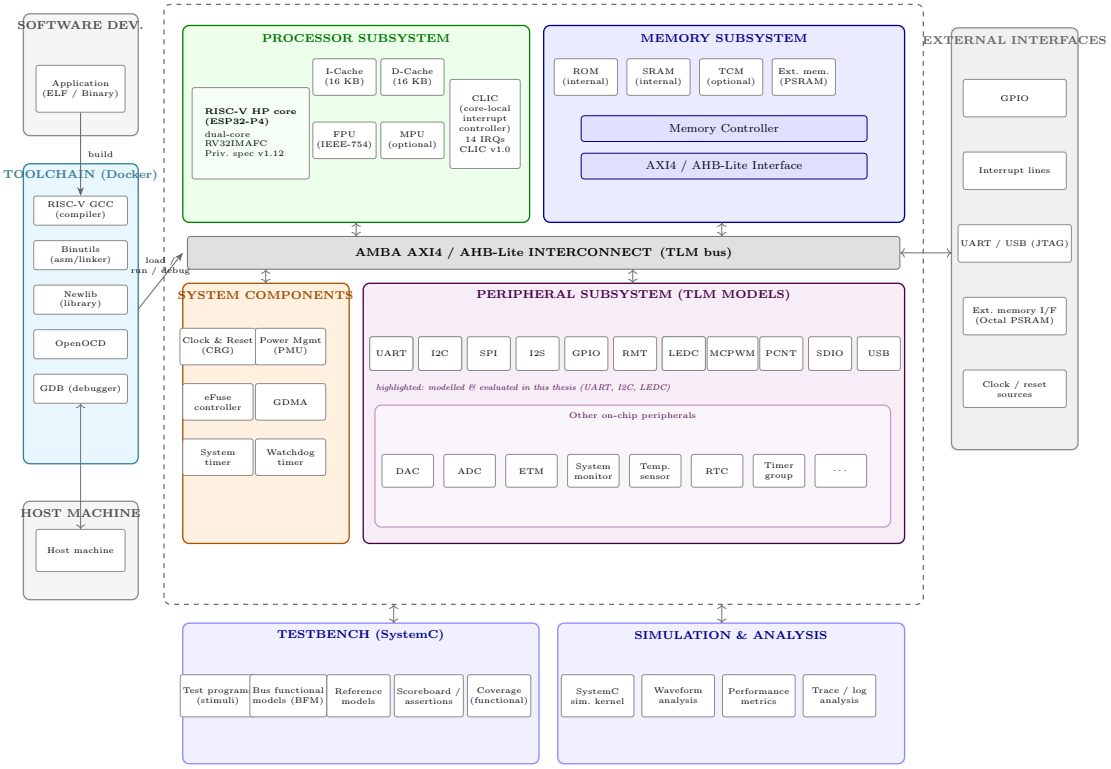


Figure 2.2: Architecture of the RISC-V virtual platform targeting the ESP32-P4.

2.2 SystemC and Transaction-Level Modelling

Because a system-on-chip contains many components—buses, memories, and processor cores—that all operate at the same time, modelling it requires a language that can express concurrency. SystemC provides exactly this: it is a C++ class library, standardised as IEEE 1666 [20], that adds a simulation kernel for concurrent processes together with hardware concepts such as modules, ports, and signals [2]. A SystemC design is a hierarchy of modules connected by channels, mirroring the way physical hardware blocks are wired together. Figure 2.3 shows a minimal example in which a CPU module and a memory module, both `sc_modules`, are connected through `sc_signal` channels inside a single top-level module.

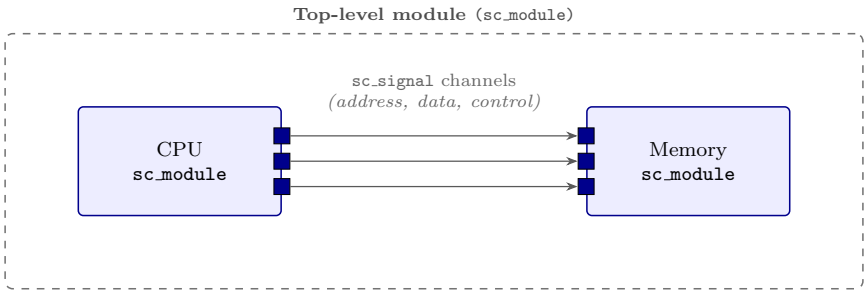


Figure 2.3: SystemC structural hierarchy: a CPU and a memory module connected by `sc_signal` channels inside a top-level module.

Connecting modules with explicit signals reproduces hardware faithfully, but it is expensive: the kernel must update every pin on every clock edge, which makes the simulation far too slow to boot an operating system. Transaction-Level Modelling (TLM-2.0) [29] removes this cost by raising the level of abstraction. Instead of toggling individual wires, a module calls a function on its neighbour and passes a single *transaction* object that carries the address, the data, and the command (read or write). Figure 2.4 contrasts the two styles. Because one function call replaces thousands of signal updates, transaction-level models are reported to run roughly one to two orders of magnitude faster than the equivalent RTL description [1]. TLM-2.0 also defines coding styles—loosely-timed, for fast software validation, and approximately-timed, when bus timing matters—so the same framework can trade speed for accuracy as required.

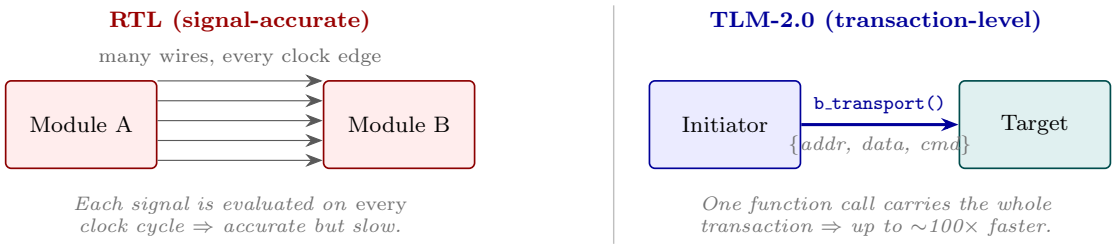


Figure 2.4: Signal-level (RTL) versus transaction-level (TLM-2.0) communication.

2.3 Register Modelling and SystemRDL

Even with the speed of transaction-level modelling, a complete virtual platform still has to describe hundreds of configuration registers, one set for every peripheral. Writing these register maps by hand in C++ is repetitive and error-prone: a single wrong offset or access permission introduces a bug that is very hard to find later. To avoid this, the register layout can be captured in SystemRDL, a register-description language standardised by Accellera [6]. SystemRDL provides a clean, machine-readable syntax for register offsets,

bit-field layouts, reset values, and read/write permissions. As shown in Figure 2.5, such a description can be processed automatically to generate the registers, their documentation, and the SystemC/TLM skeleton code—the robust, automated foundation for the generation pipeline presented in the next chapter.

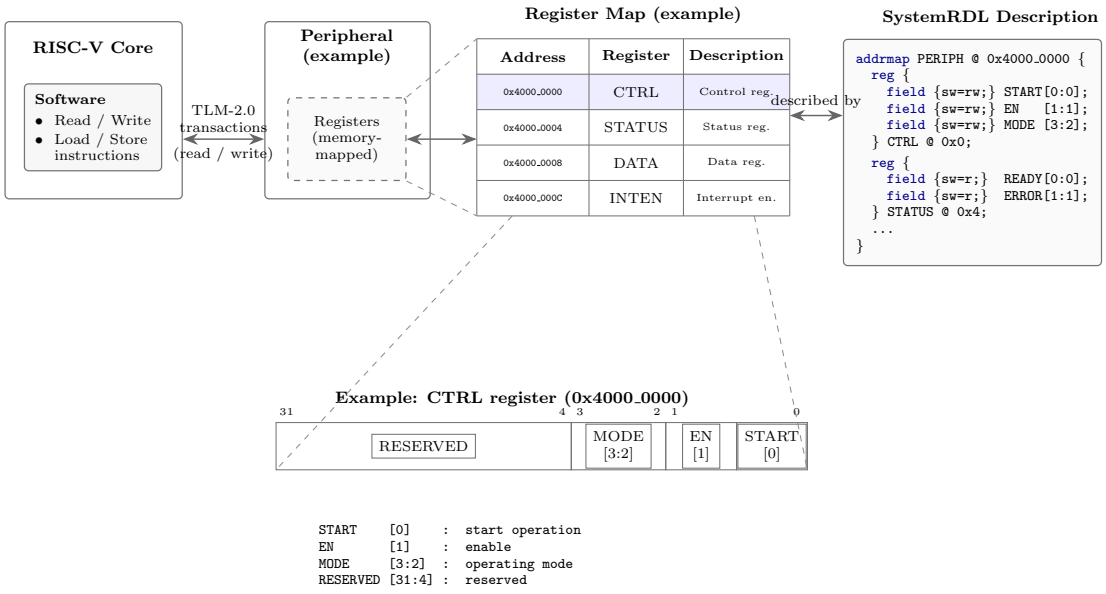


Figure 2.5: Register organisation and the corresponding SystemRDL description for a RISC-V based peripheral.

2.4 Conclusion

This chapter introduced the three foundations on which the rest of the thesis builds. It first presented virtual platforms and the abstraction levels at which hardware can be modelled, showing that the transaction level offers the best trade-off between simulation speed and accuracy, and introduced the Espressif ESP32-P4—driven entirely through memory-mapped registers—as the case study. It then described SystemC and TLM-2.0 as the framework used to build fast, executable models, and finally SystemRDL as the standardised, machine-readable register-description language that bridges hardware documentation and the generated code. Together, these elements explain why faithful register models are the cornerstone of the whole platform and set the stage for the automated generation pipeline presented in the next chapter.

3

Automated Extraction and SystemC/TLM Generation

This chapter presents the automated pipeline developed to extract hardware peripheral descriptions from Technical Reference Manuals (TRMs) and to generate SystemRDL [6] and SystemC/TLM [20, 29] models using Large Language Models (LLMs). The chapter follows the pipeline from its overview (Section 3.1) through the individual stages, and ends with the evaluation on the UART, I2C, and LEDC peripherals of the ESP32-P4 [5] (Section 3.8).

3.1 Automation Pipeline Overview

The pipeline transforms unstructured TRM documentation into SystemRDL descriptions that are validated against the SystemRDL compiler (guaranteed to compile without syntax or structural errors) and then into functional SystemC/TLM peripheral modules. It was designed as a sequence of independent, modular stages so that any component (such as the LLM backend, the compiler, or the code generator) can be replaced without affecting the rest of the workflow. The complete pipeline runs from a single entry point (`run_pipeline.py`) and executes the following six stages in order, as illustrated in the block diagram in Figure 3.1.

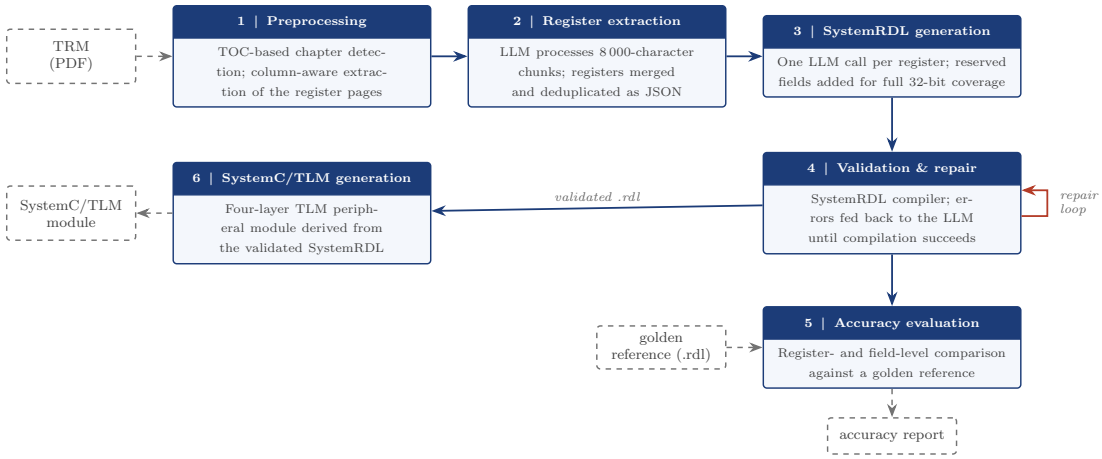


Figure 3.1: Block diagram of the six-stage pipeline from TRM PDF to SystemC/TLM.

As detailed in Figure 3.1, the automation process begins by extracting raw register data from the input TRM PDF. This data is fed into a Large Language Model (Gemini 1.5 Pro), which structures the unstructured text into a standardized JSON format. The JSON is subsequently translated into a SystemRDL description. To guarantee structural accuracy, this code enters a validation and correction loop utilizing the SystemRDL Compiler. Upon successful validation, the compiler exports an intermediate XML representation, which is finally processed by a Jinja2 templating engine to generate the executable SystemC/TLM hardware model.

Pipeline Stages

Stage 1 — Preprocessing.
A dedicated script uses the PDF's embedded Table of Contents (TOC) to locate the target peripheral chapter and isolate the register section. The selected pages are converted into structured Markdown using a column-aware extraction strategy that preserves the register table layout.

Stage 2 — Register Extraction.
The Markdown section is divided into chunks of approximately 8,000 characters. Each chunk is processed independently by the LLM, which identifies register boundaries and extracts all register fields into a structured JSON representation. Results from all chunks are merged and deduplicated by register name.

Stage 3 — SystemRDL Generation.
Each extracted register is submitted to the LLM individually. The model generates a complete SystemRDL `reg` block, including reserved fields required to cover all unassigned bit positions.

Stage 4 — Validation and Repair.
The generated SystemRDL file is compiled using the open-source SystemRDL compiler [30]. Compilation errors are fed back to the LLM together with the faulty register block, and the corrected output is reinserted until the file compiles.

Stage 5 — Accuracy Evaluation.
The validated SystemRDL description is compared against a manually created golden reference, at register and field level, producing detailed accuracy metrics. This stage runs as a side branch on the validated SystemRDL and does not gate Stage 6.

Stage 6 — SystemC/TLM Generation.
The validated SystemRDL model serves as input for automatic generation of a complete SystemC/TLM peripheral implementation ready for integration into a virtual platform.

3.2 TRM Document Preprocessing

TRM documents contain complex layouts and embedded register tables that introduce significant noise during direct PDF parsing. The pipeline therefore uses the PDF Table of Contents to locate the target peripheral chapter and keeps only the pages containing actual register tables. Reducing this surrounding noise matters because every irrelevant paragraph consumes part of the model's limited context window and increases the chance of misreading a description as a register field. The full workflow — chapter localisation, layout-aware extraction, and filtering of non-register pages — is shown in Figure 3.2; the implementation is based on PyMuPDF [31].

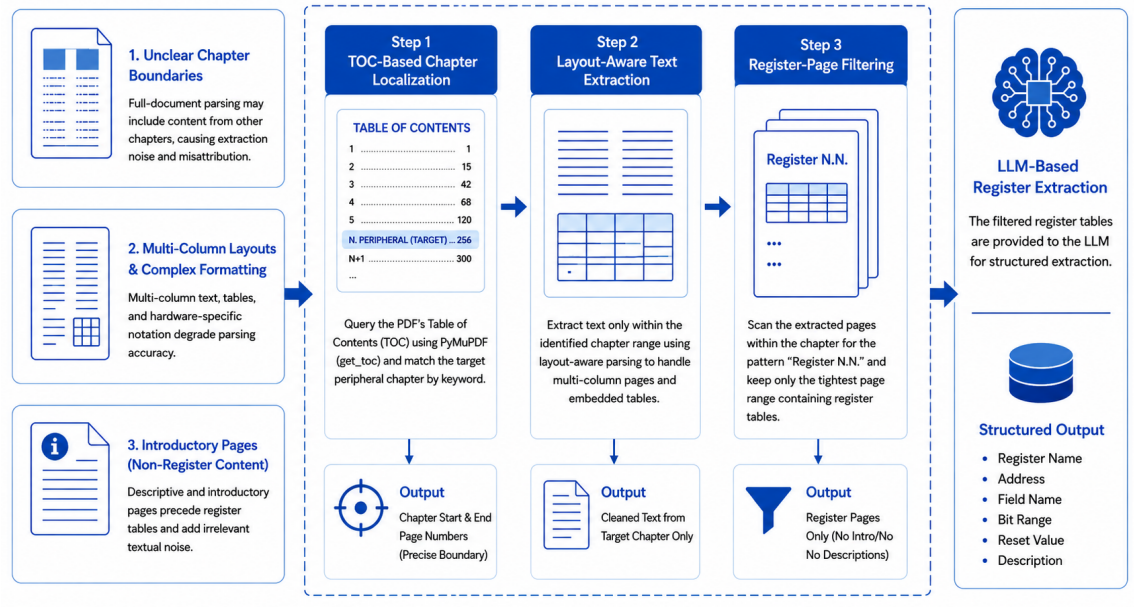


Figure 3.2: PDF preprocessing pipeline for TRM register extraction.

TOC-Based Chapter Detection

The algorithm searches the TOC for the target chapter title and determines its end boundary using the first subsequent entry at the same or higher hierarchy level. Listing 4.11 shows the core detection logic.

```
for idx, entry in enumerate(toc):
    if keyword.lower() in entry.title.lower():
        start = entry.page          # first page of target chapter

        for subsequent in toc[idx + 1:]:
            if subsequent.level <= entry.level:
                end = subsequent.page - 1   # last page before next chapter
                break

        return PageSpan(start=start, end=end)
```

Listing 3.3: TOC-based chapter detection.

Column-Aware Table Extraction

After locating the chapter, the register tables must be reconstructed from the raw PDF text. Because a PDF stores each piece of text with its position on the page rather than as structured rows, a column-aware strategy is used: the text fragments on each line are sorted from left to right, and wherever a wide horizontal gap appears, a tab stop is inserted to mark a new column (Listing 3.4). This rebuilds the original table layout and preserves the link between each register name, its bit ranges, reset values, and description. Finally, all register pages are merged into one continuous text stream, so that a register description split across two pages is not fragmented.

```
# Collect all text spans on the line with their x-positions
spans = [(s["origin"][0], s["text"])
         for s in pdf_line["spans"]]

spans.sort(key=lambda s: s[0])    # left-to-right ordering

for x, text in spans:
    if gap_exceeds_threshold(x):  # large gap => new column
        parts.append("\t")
    parts.append(text)
```

Listing 3.4: Column-aware text extraction.

3.3 LLM-Based Register Extraction

Chunking Strategy

A peripheral chapter such as the UART section spans approximately 30,000 characters after Markdown conversion. Submitting the entire section in a single call gave poor results in early experiments: only 5 of 37 registers were returned, because transformer-based models [32] lose track of content from earlier sections when the input grows too long

— registers in the second half of the chapter were skipped or had their fields merged with adjacent ones. The chapter is therefore split into chunks of approximately 8,000 characters, always cutting at blank-line boundaries so that no register description is divided across two chunks (Listing 3.6). Each chunk is processed in a separate LLM call, and the results are merged and deduplicated. Figure 3.5 illustrates the difference between the two approaches: in the single-call case the model overflows its context and recovers only the first few registers, whereas splitting the chapter into sequential chunks lets it process each part in turn and recover all registers.

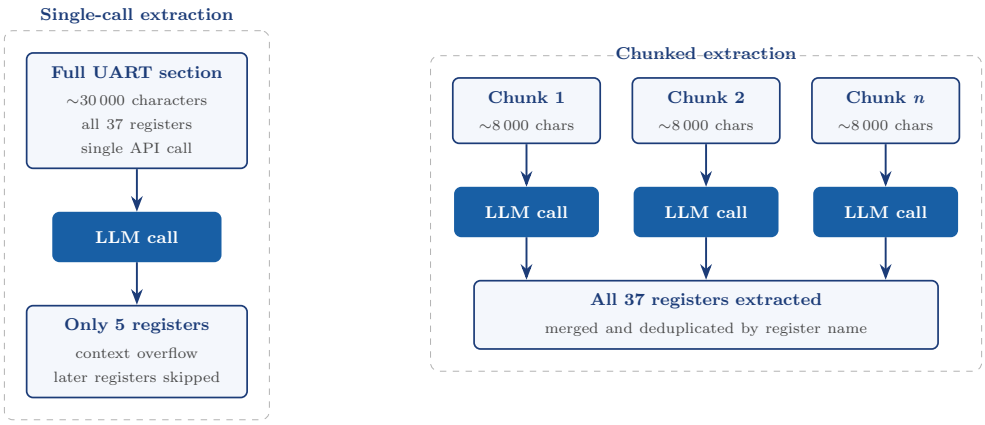


Figure 3.5: Single-call extraction (left) versus chunked extraction (right).

```python
def split_into_chunks(text: str, chunk_size: int) -> list:
    chunks = []
    start = 0
    while start < len(text):
        end = start + chunk_size
        if end < len(text):
            # Find the last blank line before the hard limit
            cutpoint = text.rfind("\n\n", start, end)
            if cutpoint != -1:
                end = cutpoint   # cut at paragraph boundary, not
                    mid-register
        chunks.append(text[start:end].strip())
        start = end
    return [c for c in chunks if c]
```

Listing 3.6: Chunk splitting at blank-line boundaries.

The function therefore searches backwards from the hard character limit for the last blank line Listing 3.6 and cuts there, so each chunk ends at a paragraph boundary rather than in the middle of a register.

Structured JSON Extraction

Each chunk is submitted with a structured prompt that instructs the model to identify all registers and return a JSON array with no surrounding text. The model handles segmentation and extraction in a single call, using surrounding context — section headings, cross-references, prose descriptions — to resolve ambiguities that a rule-based parser would miss. For each register, the model extracts the name and description, the offset (preserved exactly, including parametric formulas such as `0x0010+0x14*n` for arrayed registers), the field names with their `msb:lsb` bit ranges, the per-field access permissions, and the reset values copied verbatim including Verilog-style literals such as `9'h008`. Listing 3.7 shows an example.

```
{
  "name": "UART_INT_ENA_REG",
  "offset": "0x000C",
  "fields": [
    {
      "name": "UART_RXFIFO_FULL_INT_ENA",
      "bits": "0:0",
      "access": "rw"
    }
  ]
}
```

Listing 3.7: Example JSON output.

Deterministic Decoding with Temperature 0

The model temperature was set to 0 for all extraction and generation calls: the model then always selects the highest-probability token, so identical inputs produce identical outputs. For a hardware specification task, where any deviation from the source document is an error, this determinism is essential — a non-zero temperature could change field names, bit ranges, or reset values between runs, and it would make debugging and prompt optimisation unreliable. Figure 3.8 illustrates the difference between deterministic ($T = 0$) and stochastic ($T > 0$) decoding.

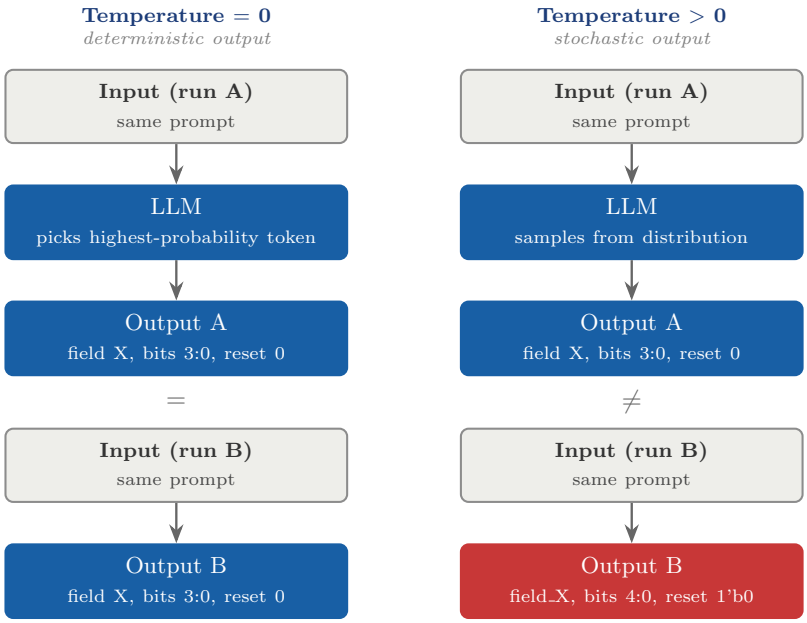


Figure 3.8: Effect of the temperature parameter on deterministic LLM inference.

Retry Mechanism and Error Isolation

Even with a careful prompt and temperature `0`, transient failures occur: API rate-limit errors from the Groq free tier, and structural JSON errors such as a missing key or a malformed bit range. Rate-limit errors are handled transparently with exponential backoff. Structural errors trigger up to three retries in which the validation errors are fed back to the model; if all retries fail, the register is written to a dedicated failure directory and the pipeline continues, so a single problematic register never blocks or corrupts the rest of the extraction.

Access Type Normalisation

Access permissions are encoded in several equivalent ways across hardware documentation — `"ro"` and `"wo"` are perfectly standard, alongside spellings such as `"read-only"` or `"read/write"` — and the LLM reproduced this variety inconsistently from one register to the next. Because the SystemRDL generator expects a single canonical vocabulary (`"r"`, `"w"`, `"rw"`), a normalisation step applied immediately after JSON parsing maps all of these variants to their canonical form (Listing 3.9), ensuring uniform input to the downstream stage regardless of which spelling the model chose.

```python
ACCESS_MAP = {
    "ro": "r",  "read-only": "r",  "read_only": "r",  "RO": "r",
    "wo": "w",  "write-only": "w", "write_only": "w", "WO": "w",
    "rw": "rw", "read-write": "rw","read/write": "rw","RW": "rw",
}

def normalize_access(reg: dict) -> dict:
    for field in reg.get("fields", []):
        raw = field.get("access", "").strip()
        field["access"] = ACCESS_MAP.get(raw, raw)  # identity if unknown
    return reg
```

Listing 3.9: Access type normalisation.

3.4 Model Selection: LLaMA 3.3 70B on Groq

Five LLM configurations were evaluated across the two pipeline stages: LLaMA 3.3 70B [33], GPT-4o [34], Mistral 7B [35], Qwen2 7B [36], and TinyLlama 1.1B [37]. The models were selected to cover a range of parameter scales, deployment environments, and architectural families. Table 3.10 summarises the comparison.

Why LLaMA 3.3 70B? LLaMA 3.3 is an open-weight model released by Meta under a permissive licence, which means it can be used, studied, and reproduced without commercial restrictions. This matters for research reproducibility: any reader can re-run the pipeline using the same model weights without requiring a paid subscription. At the 70-billion parameter scale the model has sufficient capacity to hold the structural rules of the hardware register format in its working context simultaneously with the actual register content being parsed. Smaller models at the 1B–7B range were found to lose instruction compliance as register descriptions grew longer: they would follow the JSON format rule for the first two or three fields, then begin adding free-text commentary, omitting the `access` key, or silently switching to a different field naming convention. The 70B model maintained format compliance consistently across all registers in all evaluated peripherals.

Why Groq rather than local inference? The Groq inference platform builds custom Language Processing Units (LPUs) [38] that deliver significantly lower latency per token than GPU-based inference on consumer hardware. This made the 37-register extraction run in under two minutes rather than the ten to fifteen minutes observed with local GPU inference, which is practically important when iterating on prompt improvements. Groq's free tier provided sufficient capacity for the full pipeline evaluation once a daily token budget of 100,000 tokens was managed through batching.

Why not GPT-4o? Although GPT-4o demonstrated competitive performance during preliminary experiments, it is a commercial API with usage-based pricing. Since one objective of this work is to provide a reproducible and accessible pipeline, the evaluation was conducted using freely available models hosted on Groq.

Why not smaller local models? TinyLlama in particular hallucinated field names that did not exist in the source document, presumably by interpolating from its training-data knowledge of similar peripherals. This is a critical failure mode for hardware modelling, where an incorrectly named or invented field would propagate silently through all downstream stages and produce a functionally incorrect peripheral model. Qwen2 and Mistral produced inconsistent reset values and occasionally dropped the `access` key on dense registers.

Model	Platform	Scale	Quality / Consistency	Key limitation
Groq LLaMA 3.3	Cloud (Groq)	Large (70B)	High / High	Daily token quota
TinyLlama	Local	Small (1.1B)	Low / Low	Hallucinated fields
Qwen2	Local	Medium (7B)	Medium / Medium	Inconsistent resets
Mistral	Local	Medium (7B)	Med–High / Med	High RAM usage
OpenAI GPT-4o	Cloud (OpenAI)	Large	Very High / High	API cost

Table 3.10: Comparison of the five evaluated LLM models for register extraction.

3.5 Prompt Engineering

Prompt design had a decisive influence on output quality. Three families of prompts were developed and refined iteratively based on observed failure modes.

Extraction Prompt Design

The extraction prompt constrains the model to valid JSON output with no surrounding text, enumerates all required fields, restricts access types to the vocabulary `"r"`, `"w"`, `"rw"`, and includes formatted examples taken from the target TRM. Each chunk is processed as a whole rather than register by register, so the model can use cross-register context.

Most of the prompt engineering effort addressed bit range extraction. In the vertical layout produced by the PDF extractor, each field appears as a sequence of lines — field name, reset value, MSB, LSB — and early prompt versions misread the reset value as a bit number, producing systematic one-position shifts. The final prompt explicitly annotates this line ordering (Listing 3.11), forbids assigning bit positions sequentially from zero, and requires parametric offsets such as `0x0010+0x14*n` to be preserved exactly.

```
Each field entry appears as:
  FIELD_NAME
  RESET_VALUE      <- this is the reset, NEVER a bit number
  MSB / LSB        <- the numbers after the reset

RULES:
- NEVER treat the reset value line as a bit number.
- NEVER assign bits sequentially from 0 -- read the printed numbers.
- Preserve offsets exactly, including formulas like 0x0010+0x14*n.
```

Listing 3.11: Key rules from the final extraction prompt (abbreviated).

SystemRDL Generation Prompt

A second prompt converts each JSON register into a SystemRDL `reg` block. It fixes the syntax rules — `sw`/`hw` access attributes, `[msb:lsb]` field declarations, `@` offset annotation — and requires `RESERVED` fields for all uncovered bit positions, ensuring the full 32-bit coverage needed for compilation. Reserved fields must be named `RESERVED_{MSB}_{LSB}` after the actual gap positions: generic names such as `RESERVED_0` caused collisions and comparison failures when a register contained several non-contiguous gaps. An example output is shown in Listing 3.12.

```
reg UART_INT_ENA_REG {
    desc = "UART interrupt enable register";

    field { sw = rw; hw = rw; reset = 1'b0;
        desc = "Enable UART_RXFIFO_FULL_INT.";
    } UART_RXFIFO_FULL_INT_ENA[0:0];

    // ... fields [1:1] through [19:19]

    field { sw = r; hw = rw; reset = 12'h0;
        desc = "Reserved.";
    } RESERVED_31_20[31:20];

} UART_INT_ENA_REG @ 0x000C;
```

Listing 3.12: Example generated SystemRDL block with reserved fields.

Repair Prompt

On compilation failure, a repair prompt combines the faulty SystemRDL block with the compiler error message and asks the model to return only the corrected block — not the full file — which keeps the request within the API token limit. All compilation errors were resolved within two repair iterations.

3.6 SystemRDL Generation and Validation

Per-Register Generation Strategy

The SystemRDL generation stage processes registers one at a time, following the workflow illustrated in Figure 3.13. Each register's JSON object is submitted in a separate API call, and the model returns a single `reg` block that is syntax-checked and appended to the output file. This per-register approach keeps each prompt well within the token limit (400–800 tokens per register), isolates failures — if one register produces invalid SystemRDL, only that register is re-submitted, as shown by the dashed retry path in the figure — and focuses the model's attention on a single bit layout, reducing cross-register confusion. With approximately 1,800 tokens per call against the Groq free tier's daily limit of 100,000 tokens, the 37-register UART peripheral required careful budget management, and a two-second inter-call delay prevented per-minute rate-limit violations.

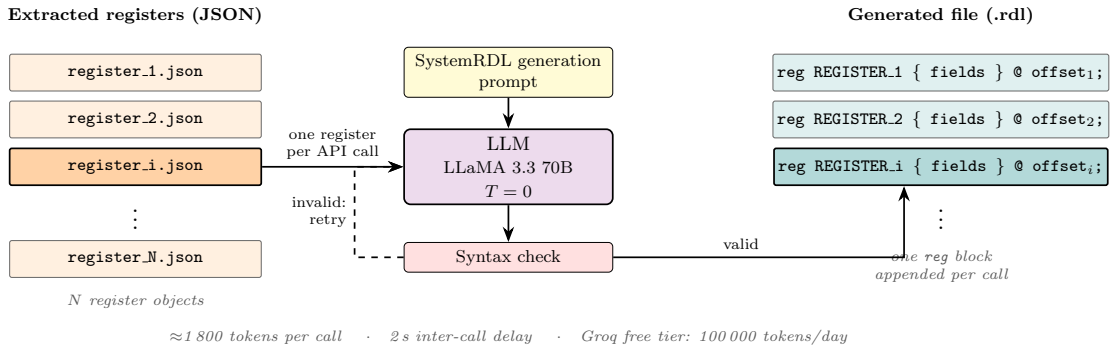


Figure 3.13: Per-register SystemRDL generation with an isolated retry path.

Compiler Validation and Repair Loop

All generated SystemRDL files were validated using the open-source SystemRDL compiler [30] after every generation and repair cycle. Figure 3.14 shows the structure of the repair loop. The most common compiler errors were a missing register name before the `@` offset annotation, field offsets exceeding the register width, the wrong property name (`access = "r"` instead of `sw = r;`), and overlapping field declarations caused by incorrect reserved field placement. On error, the compiler output was injected into a repair prompt alongside the faulty block, and the corrected version was spliced back into the file before recompilation. Table 3.15 reports the compile rates before and after this automatic repair for all evaluated models.

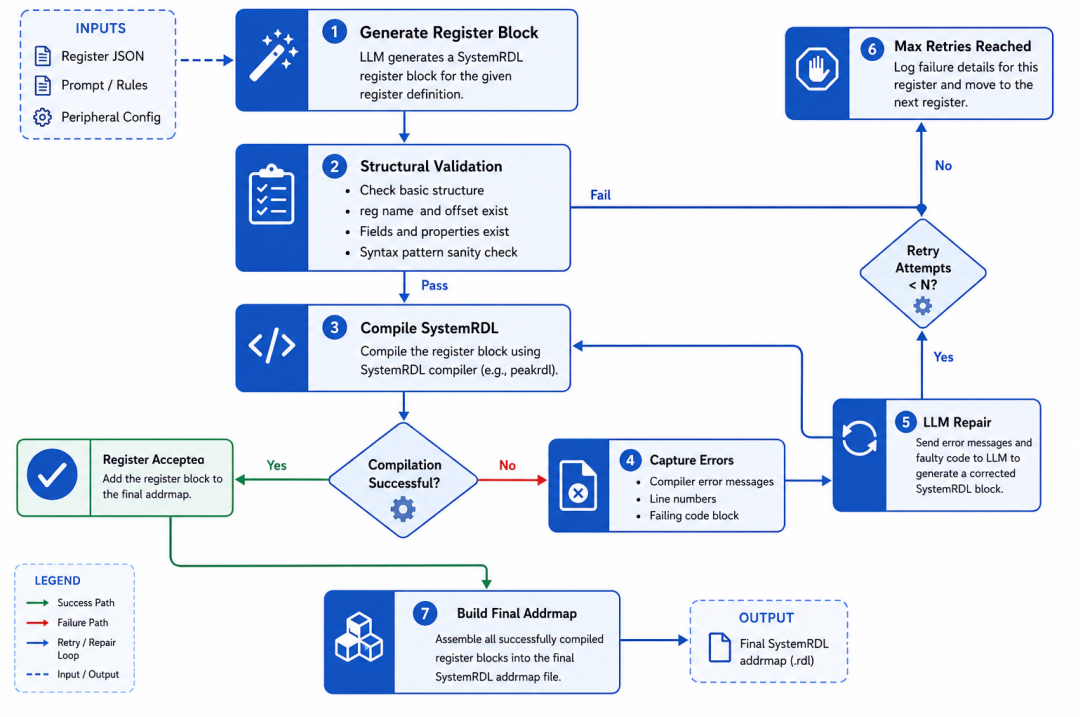


Figure 3.14: Compiler validation and LLM-based repair loop for generated SystemRDL.

Model	Params	Deployment	Licence	Context	Before (%)	After (%)	Δ (pp)
LLaMA 3.3	70B	Cloud (Groq LPU)	Open weights (Llama)	128k	91	99	+8
GPT-4o	n/a[a]	Cloud (OpenAI)	Proprietary	128k	93	99	+6
Mistral	7B	Local (CPU/GPU)	Apache 2.0	32k	70	85	+15
Qwen2	7B	Local (CPU/GPU)	Apache 2.0	32k	64	79	+15
TinyLlama	1.1B	Local (CPU)	Apache 2.0	2k	38	54	+16

[a] Parameter count not publicly disclosed.

Table 3.15: SystemRDL compiler success rates before and after automatic repair.

Two observations follow from Table 3.15. The large models already produce compilable SystemRDL for more than nine registers out of ten on the first attempt, and converge to 99% within at most two repair iterations. The improvement is largest for the smallest models (+15 to +16 pp), yet repair does not equalise them: TinyLlama remains at 54% because its failures are structural misunderstandings of the SystemRDL format, not syntax slips. The repair loop is therefore a low-cost robustness layer on top of an already capable model, not a substitute for model capability.

3.7 Automated SystemC/TLM Peripheral Generation

The final stage takes the validated SystemRDL description and generates a complete, integration-ready SystemC/TLM peripheral module. The implementation consists of three parts that mirror the generation flow: a generic hardware template (`peripheral_template.hpp`) providing the simulation infrastructure, a code generator (`generator.py`) that derives all peripheral-specific values from the SystemRDL description, and the resulting generated models (`uart_generated.hpp`, `i2c_generated.hpp`, `ledc_generated.hpp`). Every structural detail — register offsets, field widths, reset values, and access rules — is derived from the SystemRDL input, so the specification and the generated code cannot drift apart.

Base Hardware Template (`peripheral_template.hpp`)

The base component framework was developed within the Accellera SystemC IEEE 1666 standard [20] and the TLM 2.0 base protocol [29], targeting the cycle-independent simulation speed required for early firmware–hardware co-design. The template defines a class inheriting from `sc_core::sc_module`, incorporates a `tlm_utils::simple_target_socket<DerivedClass>` for transaction routing over the platform's shared bus (`SimpleBus`), and implements the standard blocking transport interface `b_transport(tlm::tlm_generic_payload&, sc_core::sc_time&)`. It is a structural blueprint: it contains no register layout of its own, but uses data-injection tokens (`{{ module_name }}`, `{{ address_mask }}`, `{{ register_cases }}`) as deterministic anchors for automated code insertion. Figure 3.16 shows the resulting module architecture. Three design decisions are engineered into the skeleton.

Parametric dynamic address truncation. Global system buses carry absolute 32-bit physical addresses. To decouple peripheral models from memory-map reallocations while supporting variable-sized IP blocks, the template avoids hardcoded boundaries in favor of a region-aware address decoding mask (M_A). The local offset is evaluated as

$$\text{Offset}_{\text{local}} = \text{Address}_{\text{bus}} \ \& \ M_A. \tag{3.1}$$

The mask width is derived per peripheral from the address allocation window in the ESP32-P4 memory map [5]: a uniform 12-bit mask would clip larger subsystems, so the generator applies $M_A = 0\text{xFFF}$ (12 bit) for standard 4 KB blocks, $M_A = 0\text{x3FFF}$ (14 bit) for the 16 KB Ethernet (EMAC) region, and $M_A = 0\text{x3FFFF}$ (18 bit) for the 256 KB aperture of the USB 2.0 OTG High-Speed engine. The aperture size matters in practice: the UART0 block must expose its full 4 KB window up to `0x500C_AFFF` — restricting it to a 1 KB boundary (`0x500C_A3FF`) triggers a Load Page Fault (`mcause: 0xd`) as soon as low-level drivers poll the upper status registers.

Decoupled interrupt linkage. Interrupt handling separates the core-private interrupt controller (CLIC) from the peripheral routing matrix. The Interrupt Matrix is bound to its physical base address `0x500D_6000` within the HP Peripherals 1 domain [5], and the template accesses it through a direct pointer (`clic* interrupt_gateway`) combined with a runtime registration routine, `setup_interrupts(unsigned int irq_`

`id`). This keeps cross-module compilation dependencies minimal while following the SoC's interrupt prioritization scheme.

Overridable write hook for debug telemetry. To observe edge-case behaviour — for example, tracking the characters a bare-metal `printf` writes into `UART_FIFO_REG` at offset `0x0` — the template declares an overridable method, `virtual void handle_register_write(uint64_t addr, uint32_t value)`, invoked inside the write transport path. Subclasses can intercept these writes and, for instance, echo them to the simulation terminal, without modifying the core transaction logic.

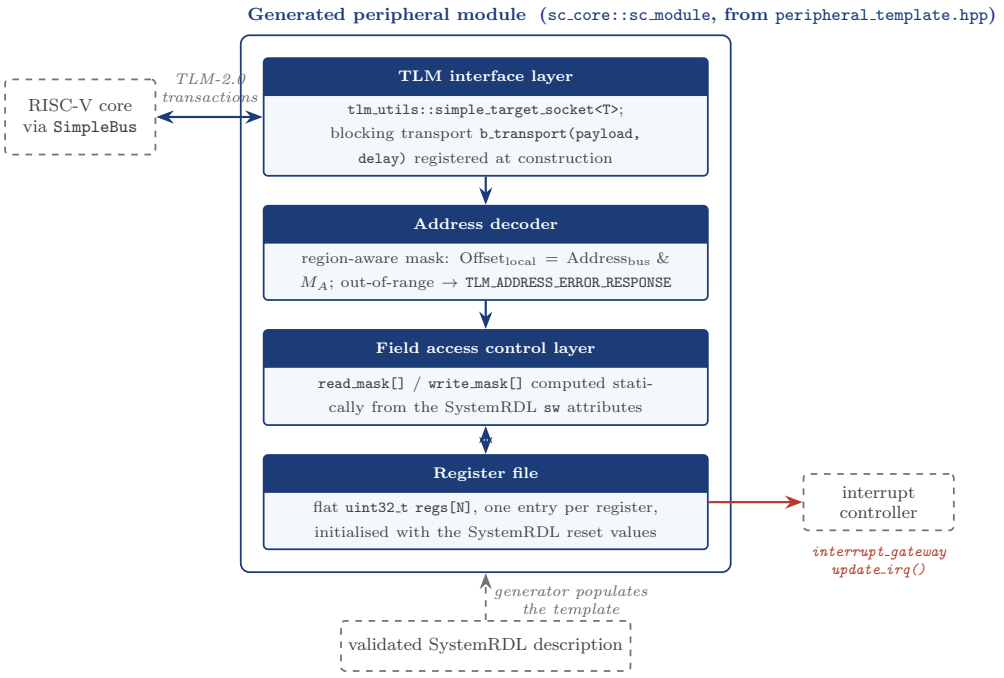


Figure 3.16: Four-layer architecture of a generated SystemC/TLM peripheral module.

Register Extraction and Bitmask Generation (`generator.py`)

The translation of the validated register description into a C++ compilation unit is handled by `generator.py`, built on top of the Accellera-compliant SystemRDL 2.0 frontend compiler [30]. The script executes a three-stage flow over the compiled specification:

1. **AST traversal and component enumeration.** The engine recursively walks the elaborated SystemRDL object hierarchy, extracting `addrmap`, `regfile`, `reg`, and `field` instances. For every register it records the relative byte offset, the identifier, and the bit-width.
2. **Computation of structural bitmasks.** A common weakness of manually written virtual prototypes is the incorrect handling of unmapped or reserved bits inside a 32-bit register. The generator addresses this by computing, for every register, an exact **write**

mask (M_W) and **read mask** (M_R) from the field declarations. For a software-writable field spanning the slice $[B_{\text{high}} : B_{\text{low}}]$, the contribution to the write mask is

$$M_W = \sum_{i=B_{\text{low}}}^{B_{\text{high}}} 2^i. \tag{3.2}$$

Bits belonging to reserved ranges are forced to zero in the mask, so bus transactions targeting unmapped bit positions are nullified before they can alter the internal state.

3. **Template rendering.** To keep the Python code independent of any particular peripheral, the computed metadata (masks, offsets, identifiers, reset vectors) is assembled into a data dictionary and passed to the Jinja2 template engine [39], which substitutes the placeholders of `peripheral_template.hpp` with the corresponding C++ code blocks.

Generated Peripheral Models (`*_generated.hpp`)

The generator outputs standalone header units — `uart_generated.hpp`, `i2c_generated.hpp`, and `ledc_generated.hpp` — containing the concrete C++ classes that implement the register maps of the ESP32-P4 TRM [5]. Their fidelity to the silicon rests on three mechanisms.

TRM-defined reset vectors and base offsets. Each generated constructor seeds the internal storage with the reset constants from the SystemRDL description, and the platform configuration layer maps every block to its official base address. This corrected an error inherited from older address maps, which placed the LED PWM Controller at `0x5001_9000` — inside the USB 2.0 OTG range. The generated platform maps LEDC to its official base `0x500D_3000` in the HP PERI1 domain, with the Interrupt Matrix at `0x500D_6000` [5].

Transaction dispatch via a per-offset `switch`. The transport method routes the truncated local address through a dense, compiler-friendly `switch(addr)` layout in which each valid register offset matches one `case` block. A transaction targeting any address outside the register map falls through to the `default:` branch and returns `tlm::TLM_ADDRESS_ERROR_RESPONSE`, so misaligned or stray firmware accesses can never silently corrupt simulator state.

Mask-guarded register writes. Inside each write case, the payload extracted from the TLM generic interface is applied with a clear-and-set sequence using the precomputed write mask of Equation (3.2): writable bits are replaced, read-only and reserved bits are preserved. Listing 3.17 shows the generated pattern.

```
void b_transport(tlm::tlm_generic_payload& trans, sc_time& delay) {
    // Truncation with the region-aware mask injected by generator.py
    uint64_t addr  = trans.get_address() & 0xFFF;     // M_A, 4 KB block
    uint8_t* ptr   = trans.get_data_ptr();
    bool     is_wr = (trans.get_command() == tlm::TLM_WRITE_COMMAND);

    switch (addr) {
    case 0x000C: {                                    // UART_INT_ENA_REG
```

```
        if (is_wr) {
            uint32_t w = *reinterpret_cast<uint32_t*>(ptr);
            // clear-and-set with the write mask M_W
            int_ena = (int_ena & ~0x000FFFFFu) | (w & 0x000FFFFFu);
            handle_register_write(addr, w);           // debug hook
            update_irq();
        } else {
            *reinterpret_cast<uint32_t*>(ptr) = int_ena;
        }
        break;
    }
    // ... one case per register, generated from the SystemRDL map ...
    default:                                          // unmapped offset
        trans.set_response_status(tlm::TLM_ADDRESS_ERROR_RESPONSE);
        return;
    }
    delay = sc_time(10, SC_NS);
    trans.set_response_status(tlm::TLM_OK_RESPONSE);
}
```

Listing 3.17: Generated `b_transport()` structure: region-aware address truncation (Equation (3.1)), per-offset `switch` dispatch, and mask-guarded writes (Equation (3.2)). One `case` block is emitted per register from the SystemRDL address map.

Interrupt Interface

For peripherals with interrupt enable and status registers, the generated model recomputes its interrupt condition after every write transaction through an `update_irq()` function (Listing 3.18). Rather than exposing a standard SystemC `sc_out<bool>` port, the module raises the interrupt through the direct pointer to the framework's Core-Local Interrupt Controller (`interrupt_gateway`) described in Section 3.7, calling its registration routine at construction so the peripheral is bound to the correct interrupt line of the Interrupt Matrix. This avoids the overhead of signal-port binding and matches the inheritance-based CLIC model used by the virtual platform.

```
void update_irq() {
    uint32_t active = int_ena & int_st;   // enabled AND pending
    if (active != 0)
        interrupt_gateway->trigger(irq_id);  // raise via the CLIC pointer
}
```

Listing 3.18: Interrupt output logic.

Integration into the Virtual Platform

The three generated modules were integrated into the RISC-V virtual platform targeting the ESP32-P4, at the base addresses specified in the TRM memory map [5]: UART0

at 0x500C_A000 (full 4 KB aperture), I2C at 0x500C_4000, and LEDC at 0x500D_3000. Functional simulation confirmed correct read/write behaviour, correct reset values at construction, rejection of out-of-range accesses, and interrupt outputs toggling in response to writes to the enable and status registers. Table 3.19 compares the three generated modules.

Property	UART	I2C	LEDC
Total registers	37	35	19
Non-zero reset registers	8	11	2
Interrupt enable register	`UART_INT_ENA_REG`	`I2C_INT_ENA_REG`	`LEDC_INT_ENA_REG`
Interrupt status register	`UART_INT_ST_REG`	`I2C_INT_STATUS_REG`	`LEDC_INT_ST_REG`
Address mask M_A	0xFFF (4 KB)	0xFFF (4 KB)	0xFFF (4 KB)
Integration base address	`0x500C_A000`	`0x500C_4000`	`0x500D_3000`
Simulation result	Correct R/W + IRQ	Correct R/W + IRQ	Correct R/W + IRQ

Table 3.19: Comparison of the generated UART, I2C, and LEDC modules.

3.8 Experimental Results and Discussion

This section evaluates the pipeline on three peripherals of the ESP32-P4 TRM [5] — UART, I2C, and LEDC — using Groq LLaMA 3.3 70B as the primary model. The three were selected because they exercise different difficulty profiles: UART has a large register set with dense single-bit interrupt fields, I2C contains timing registers whose reset values are defined outside the register tables, and LEDC makes heavy use of arrayed channel and timer registers with parametric offsets.

Evaluation Methodology

Each generated SystemRDL description is compared field-by-field against a manually created golden reference derived from the TRM, on seven metrics: register identification, register name accuracy, offset correctness, field name accuracy, bit range correctness, access type accuracy, and reset value accuracy. Two normalisation rules avoid penalising cosmetic differences: reserved fields are matched by bit range rather than by name (a generated `RESERVED_0` covering `[31:20]` is equivalent to the golden `RESERVED_31_20`), and offsets and reset literals are compared after whitespace removal, case folding, and radix-aware value comparison, so that `1'b0`, `0x0`, and `0` compare equal. Since all calls use temperature 0, every reported number comes from a single deterministic run.

Overall Accuracy

Table 3.20 summarises the top-level results. All 37 UART registers, all 35 I2C registers, and all 19 LEDC registers were correctly identified and extracted; no register was missed,

duplicated, or incorrectly named.

Peripheral	Registers	Extracted	Fields (%)	Reset val. (%)	Overall (%)
UART	37	37/37 (100%)	100	97.8	99.6
I2C	35	35/35 (100%)	98.3	92.4	96.6
LEDC	19	19/19 (100%)	93.0	100	94.1

Table 3.20: Per-peripheral extraction results using Groq LLaMA 3.3 70B.

Detailed Per-Metric Comparison

Table 3.21 breaks the results down across all seven metrics. Two patterns emerge. First, purely *structural* information — register identification, naming, field names, bit ranges, and access types — is extracted with very high reliability on all three peripherals. Second, each peripheral exposes one characteristic weakness tied to how the TRM encodes the corresponding information: reset values for I2C, and offsets for LEDC. Both are analysed in the discussion below.

Metric	UART (37 reg.)	I2C (35 reg.)	LEDC (19 reg.)
Register identification (% found)	100	100	100
Register name accuracy (%)	100	100	100
Offset correctness (%)	100	97.1	79.0
Field name accuracy (%)	100	98.3	93.0
Bit range correctness (%)	99.6	97.1	89.0
Access type accuracy (%)	99.6	98.6	98.0
Reset value accuracy (%)	97.8	92.4	100
JSON validity rate (%)	100	100	98.0
SystemRDL compile rate after repair (%)	99	99	97

Table 3.21: Detailed per-metric extraction results across the three peripherals.

Discussion

UART (99.6% overall). UART is the best case for the pipeline: a long but regularly structured chapter with fixed offsets. All structural metrics reached 100%; the only errors were two reset values. `UART_CLKDIV_SYNC_REG` expresses its reset as a clock-dependent formula that no text-level extraction can resolve into a single constant, and one field of `UART_CONF0_SYNC_REG` has its reset documented in a footnote that lands on a different page after PDF conversion, outside the chunk seen by the model. Both errors are attributable to the source document's encoding, not to model behaviour.

I2C (96.6% overall). The I2C score is dominated by a single metric, reset value accuracy at 92.4%. A handful of timing configuration registers reset to protocol-specific delay constants that the TRM defines in a timing-diagram section, not in the register tables. Since the extraction stage deliberately isolates the register pages, the model sees no reset information for these fields and defaults to zero. All remaining metrics stayed at or above 97%, confirming that the weakness is informational — the value is simply absent from the model's input — rather than a parsing failure. A future extension could detect such cross-references and selectively include the referenced section in the prompt.

LEDC (94.1% overall). LEDC exposes the third failure class — parametric offset notation. Its results break down as follows:

- Much of the LEDC register map is arrayed: per-channel registers (`LEDC_CHn_CONF0_REG`) and per-timer registers (`LEDC_TIMERn_CONF_REG`) are documented once with a stride formula such as `0x0000+0x14*n`.
- The TRM writes these formulas inconsistently (spacing, multiplication symbol, operand order, radix), and column-aware PDF extraction sometimes splits them across spans. The model therefore reproduced several offsets that are numerically correct but textually different, and occasionally dropped the stride term.
- Normalisation absorbs the cosmetic variants; the residual 21% are genuinely altered or truncated formulas that should not be silently accepted. Parametric offsets are thus the next target for the rule-based post-processing stage.
- The lower field-name (93.0%) and bit-range (89.0%) accuracies fall in these same arrayed groups, where the repeated table layout is fragmented by PDF conversion. Reset value accuracy reached 100%, as the LEDC chapter defines all resets directly inside the register tables.

Error taxonomy. Table 3.22 consolidates the observed errors into four root-cause classes. Notably, none of them corresponds to free hallucination: every error traces to a specific property of the source document or its PDF encoding. Errors that are explainable are errors that can be engineered away.

Error class	Affected	Root cause
Formula reset values	UART	Reset defined as a clock-dependent expression, not a constant.
Cross-page footnotes	UART	Field reset documented in a footnote separated from the register by PDF pagination.
Cross-referenced resets	I2C	Timing constants defined in a diagram section outside the register tables.
Parametric offsets	LEDC	Inconsistent stride-formula notation; formulas occasionally truncated to the base address.

Table 3.22: Taxonomy of the extraction errors observed across the peripherals.

Validation of the Generated SystemC/TLM Modules

The generated modules were exercised through TLM transactions on the RISC-V virtual platform, covering four aspects: (i) *reset correctness* — every register read back after construction and compared against the SystemRDL reset values; (ii) *read/write semantics* — writes to `rw` fields persist, writes to read-only fields are silently masked, reads of write-only fields return zero; (iii) *address decoding* — out-of-range transactions return `TLM_ADDRESS_ERROR_RESPONSE`; and (iv) *interrupt behaviour* — the `irq` output asserts exactly when enable AND status is non-zero.

All four checks passed for the UART, I2C, and LEDC modules. For I2C, the timing registers with incorrect reset values were corrected manually before integration — under five minutes of work, against an estimated full day for modelling the peripheral by hand. It must be noted that these tests were authored within this work and therefore demonstrate internal consistency between the SystemRDL description and the generated module; validation against independent third-party driver code (the ESP-IDF UART HAL layer) is the necessary next step for claiming functional correctness, and is discussed in the future-work section.

Practical impact. Across all three peripherals, the pipeline reduced the modelling effort from an estimated full working day per peripheral to a few minutes of pipeline runtime plus minutes of targeted manual correction. The use of the open-weight LLaMA 3.3 model on the free Groq tier means the complete experiment can be reproduced at zero cost, which matters for adoption in academic and resource-constrained industrial settings.

3.9 Conclusion

This chapter presented the complete automated pipeline from TRM PDF to SystemC/TLM peripheral model, combining PDF parsing, LLM-based register extraction, SystemRDL generation with compiler-guided repair, and template-based code generation using LLaMA 3.3 70B at temperature 0. Evaluated on the ESP32-P4 UART, I2C, and LEDC peripherals, it extracted structural register information with high reliability, the residual errors being confined to a few fields and traced to encoding properties of the source document. The three generated modules were integrated into the RISC-V virtual platform; their functional testing and a broader assessment of the approach are the subject of the next chapter.

4

Implementation Evaluation and Functional Verification

Co-verifying low-level bare-metal firmware against a virtual hardware model is one of the decisive steps in modern electronic system-level (ESL) design. The purpose of this chapter is to establish that the virtual peripheral models developed for the RISC-V Virtual Prototype (VP) of the Espressif ESP32-P4 are both architecturally correct and functionally faithful to the real silicon.

The evaluation proceeds in the order the work was carried out. The virtual platform was first prepared by hand, including its two most demanding components—the Core Local Interrupt Controller (CLIC) and the Image Signal Processor (ISP) (Section 4.2). That experience exposed the cost of purely manual modeling and motivated the automated generation pipeline, which is then evaluated in two complementary phases: **structural verification** of the auto-generated register peripherals (Section 4.3), and **behavioural verification** of their manually refined counterparts (Section 4.4). Figure 4.1 summarises the two phases and how their outputs converge on the virtual platform.

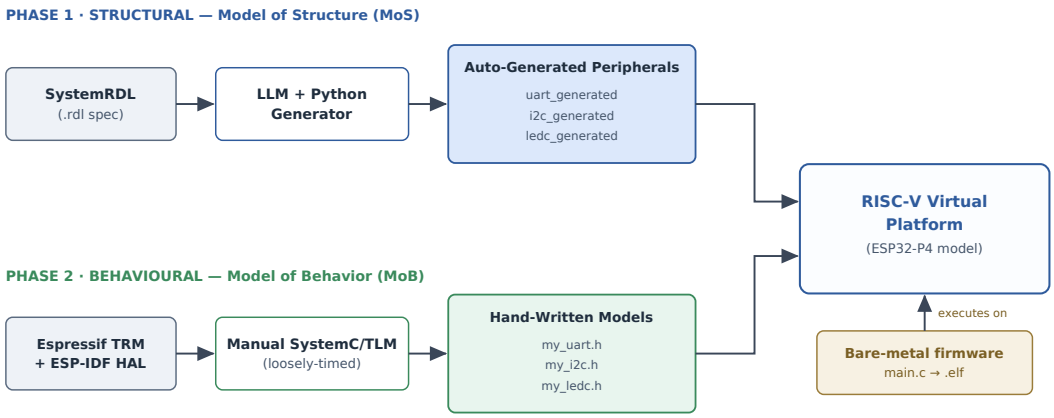


Figure 4.1: Evaluation flow: Structure and Behavior models.

4.1 Experimental Environment and Development Infrastructure

The pipeline and its evaluation were carried out inside a containerised development environment, chosen so that every result reported in this chapter can be reproduced independently of the host machine. Containerisation isolates the toolchain dependencies and guarantees identical behaviour across the different systems used during the project. The environment is built from two cooperating Docker images held in a private registry at Universität zu Lübeck (`registry.iti.uni-luebeck.de`). Splitting it along its natural fault line—hardware simulation on one side, embedded-software compilation on the other—lets each image be updated, scaled, and optimised without disturbing the other.

Container Architecture

SystemC simulation image The first image (23.1 GB, 8.47 GB compressed) carries the complete hardware-modeling stack. Built on Ubuntu 24.04, it pairs the GCC 14 and Clang 18 compilers (full C++17/20 support) with *SystemC 3.0.1*, the open-source implementation of the IEEE 1666-2023 standard that supplies the transaction-level (TLM-2.0) semantics and event-driven concurrency on which the virtual platform depends. Around this core sit the tools the workflow relies on: the CCI 1.0.1 configuration interface for parameterisable module instantiation; the CMake and Make build chain; and a debugging and analysis suite comprising GDB with remote-protocol support, Valgrind, and the `lcov`/`gcovr` coverage tools. Boost, SDL2, RapidJSON, libcurl, and libzstd provide the remaining runtime services. Rather than compiling SystemC and CCI inside the image, both are installed from pre-built artifacts—a single decision that shortens image preparation from roughly four hours of full compilation to about fifteen minutes.

RISC-V toolchain image The second image (33.1 GB, 6.6 GB compressed) provides the embedded-software side for the ESP32-P4 target. At its centre is the `riscv64-unknown-elf` GCC cross-compiler configured for the RV32IMAFC instruction set, together with the matching GNU Binutils and the Newlib C library intended for systems without an operating system. The Espressif IoT Development Framework (ESP-IDF) headers and low-level peripheral drivers—for the UART, the I2C controller, and the LEDC PWM controller—are included so that production firmware compiles unmodified, while GDB with OpenOCD support covers both on-chip and simulation-based debugging. Together these turn bare-metal C source into the ELF binaries that are subsequently executed inside the virtual platform.

Build Orchestration and Reproducibility

A Makefile-based build system hides the container details behind a single interface. It detects the available runtime automatically, preferring Docker where present and falling back to Podman otherwise, and both images are published for `amd64` and `arm64` so the environment runs equally on Intel/AMD workstations and on ARM hosts such as Apple-Silicon Macs. When a container is launched, the repository is mounted at `/root/BuddelChip`, giving the tools direct access to the source tree and generated artifacts. Reproducibility is enforced by pinning every significant component to an explicit version—Ubuntu 24.04 LTS,

GCC 14, SystemC 3.0.1, and Python 3.12, with all external library versions fixed in the Dockerfile—and by tagging images per Git branch. Combined with the multi-architecture support, these constraints ensure that a result obtained on one machine can be reproduced on any other with only Docker or Podman installed. Finally, every pipeline stage isolates its failures: if a single register fails to extract, the stage continues with those that succeeded rather than aborting, reflecting the reality that no fully automated extraction succeeds on every register.

4.2 Platform Preparation: Manual Modeling of CLIC and ISP

Before the automated pipeline existed, the virtual platform had to be brought up by hand. Most of the SoC reduces to register-mapped peripherals, but two components do not: the **Core Local Interrupt Controller (CLIC)** and the **Image Signal Processor (ISP)**. Their complexity lies in routing and dataflow logic rather than in a register layout, so they had to be modeled manually and in full. This section reports that work for two reasons. First, the CLIC and ISP are the architectural substrate on which the later auto-generated peripherals run—the generated models route their interrupts through the very CLIC described here. Second, the effort they demanded—several days per component—made the division of labour explicit (Figure 4.2) and directly motivated the automated pipeline evaluated in the remainder of the chapter.

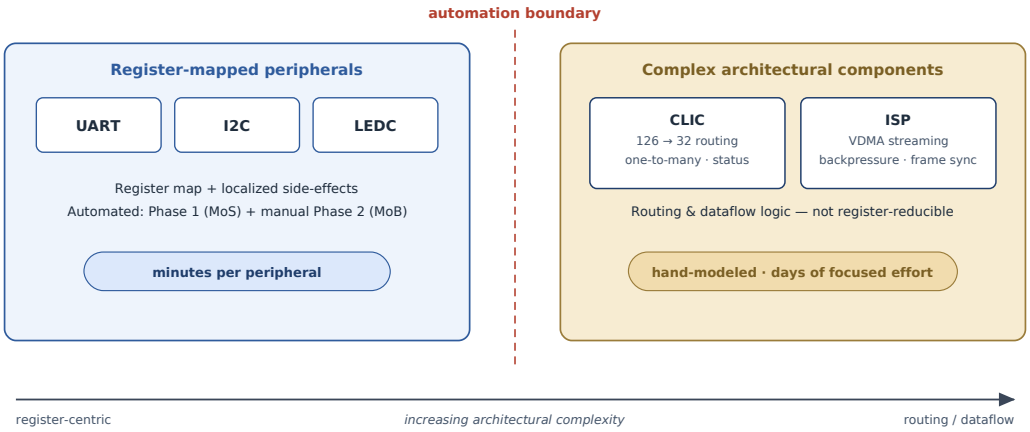


Figure 4.2: Manual modeling reveals the automation boundary.

CLIC: Interrupt Matrix Modeling

Architecture The CLIC routes 126 external interrupt sources onto 32 peripheral lines per core [5], and additionally supports one-to-many mapping, in which a single source asserts several CPU lines simultaneously. Beyond routing, the controller exposes Interrupt Status Registers [5] that report the global state of all 126 sources independently of any routing configuration. Figure 4.3 shows the functional flow.

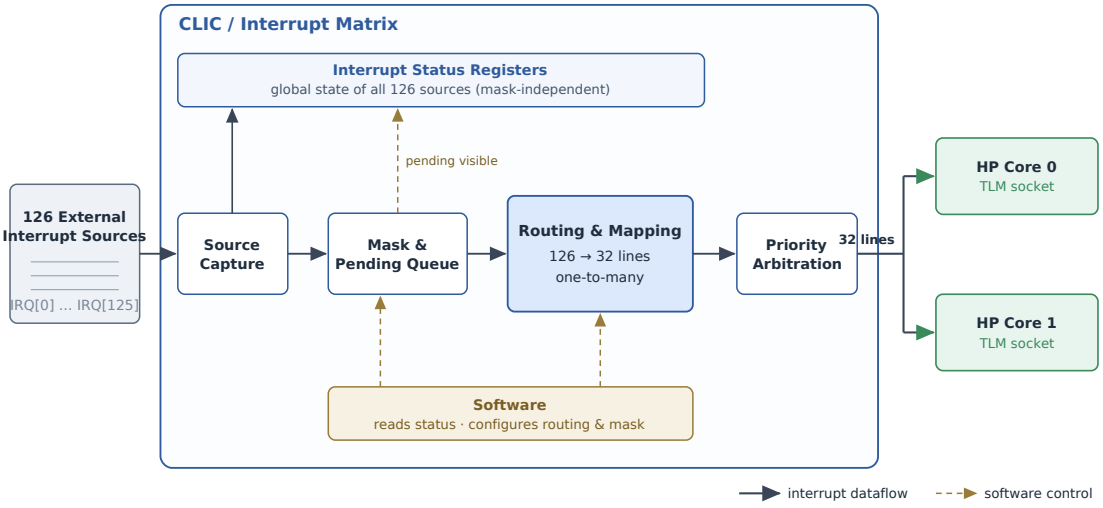


Figure 4.3: Interrupt Matrix (CLIC) functional flow.

SystemC/TLM model The model exposes the 126 inputs as `sc_signal` ports and connects to the high-performance cores through two TLM sockets. The principal challenge was state coherence between masking and observability: when software masks an interrupt the hardware queues it, yet the status registers must still report its existence. This was resolved with an internal pending queue coupled directly to the configuration registers [5]—a masked source enters the queue, a status read still reports it, and unmasking releases it back to the routing logic.

Validation The testbench exercised interrupt routing, dynamic reconfiguration, and global status reads. The worst-case scenario, termed *priority bombing*, fired ten interrupts simultaneously to confirm that the highest-priority sources reach the 32-line interface intact. Table 4.2 summarises the outcomes.

Scenario	Property checked	Result
Static routing	126 → 32 line mapping	Passed
One-to-many mapping	single source → *N* CPU lines	Passed
Dynamic reconfiguration	remapping at runtime	Passed
Global status read	all 126 sources visible while masked	Passed
Priority bombing	2 simultaneous sources; highest wins	Passed

Table 4.4: CLIC validation: testbench scenarios and outcomes.

ISP: Streaming Pipeline Modeling

Architecture The ISP is a throughput engine. Although it can capture from MIPI-CSI or DVP interfaces, this work targets the memory-to-memory mode, in which data flows through a Virtual DMA (VDMA). The VDMA presents two distinct handshaking interfaces: Interface 1 for ISP-to-memory and Interface 2 for memory-to-ISP [5]. Figure 4.5 shows the complete pipeline.

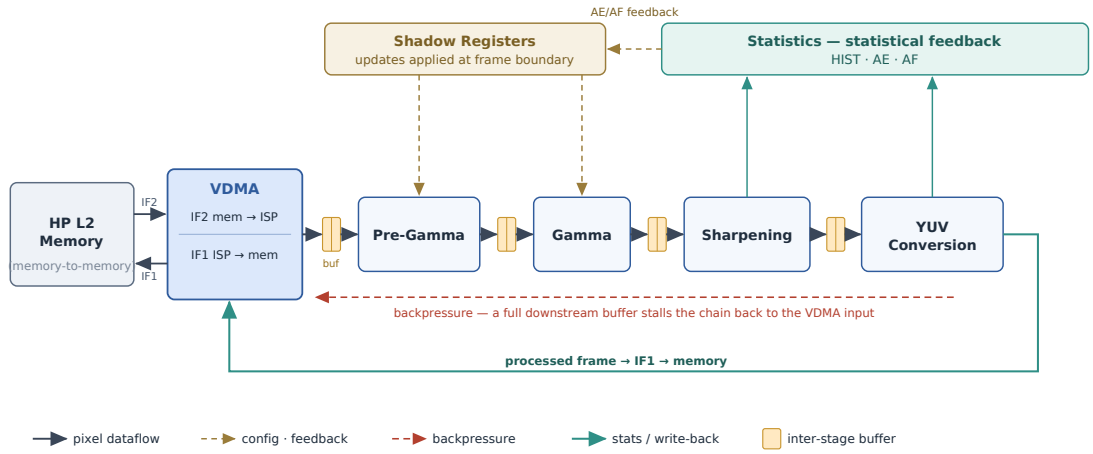


Figure 4.5: ISP processing pipeline and VDMA integration

SystemC/TLM model The model uses stage-based buffers for the pre-gamma and post-gamma stages (sharpening, YUV conversion) [5]; pixels accumulate latency as they advance through the chain. The principal challenge was pipeline synchronisation: when a downstream buffer fills, the resulting stall must propagate backward through the entire chain to the VDMA input. Reproducing this behaviour faithfully required careful modeling of the backpressure requirements described in the VDMA flow-controller section of the reference manual [5]. A further constraint is frame coherence: register updates are buffered and applied only at frame boundaries, so that mid-frame reconfiguration cannot corrupt the image.

Validation Synthetic gradient images were generated and their output values checked to ensure that transformations matched expectations across the RGB and YUV stages. Small images (160 × 120) were used to verify the pipeline logic, the backpressure stalls, and the statistical modules [5]. Table 4.2 summarises the outcomes.

Lessons: From Manual Effort to Automation

Building the CLIC and ISP by hand made one lesson explicit: the dominant cost lies in *understanding* the hardware, not in writing the code. A single misread register offset or an overlooked handshake requirement leads to hours of debugging, and the real complexity is consistently buried in the details—the post-gamma stages, the 126 → 32 mapping, the

Aspect	Check	Result
RGB stages	output vs. expected transform (BF, LSC, demosaic, CCM)	Passed
YUV conversion	output vs. expected transform (RGB2YUV, YUV2RGB)	Passed
Backpressure	downstream-full stalls the VDMA input	Passed
Frame boundaries	configuration updates applied only per frame	Passed
Statistics modules	`HIST`, `AE`, `AF`, `AWB` vs. reference	Passed

Table 4.6: ISP validation on synthetic gradient images (160 × 120).

backward-propagating stall. Each component required several days of focused work, a fragility that is a recognised bottleneck in virtual prototyping.

Crucially, doing this work first made the division of labour visible (Figure 4.2). The repetitive part—transcribing register maps—was ripe for automation, whereas the routing and dataflow logic of the CLIC and ISP was irreducibly manual. The former became the target of the pipeline evaluated in Sections 4.3–4.7; the latter remains the class of component for which a register-centric generator cannot substitute for an expert. To keep results trustworthy, the manual models were cross-checked against external emulators where possible, following established comparative evaluation practices in modern ISA simulation research.

4.3 Phase 1: Structural Evaluation of Auto-Generated Peripherals

The first phase of the evaluation utilizes a specialized Python-based generator designed to compile machine-readable register definitions directly from SystemRDL (`.rdl`) source files. This automated approach provides rapid compilation of complex register maps while ensuring absolute layout consistency, addressing the challenge of manual register transcription errors identified in Section 1.2.

Hardware Simulation Infrastructure

To evaluate the correctness of the synthesized files, a specialized hardware simulation framework was constructed using the top-level application wrapper `test.cpp`. This file acts as the primary virtual platform instantiation environment. The framework instantiates three core auto-generated structural modules: `uart_generated`, `i2c_generated`, and `ledc_generated`.

These blocks are interconnected with a target RISC-V Instruction Set Simulator (ISS) core through standard Transaction-Level Modeling (TLM 2.0) blocking transport interfaces. Incoming bus transactions are captured as generic payloads, unpacked, and routed through mapped target sockets to verify address decoding logic.

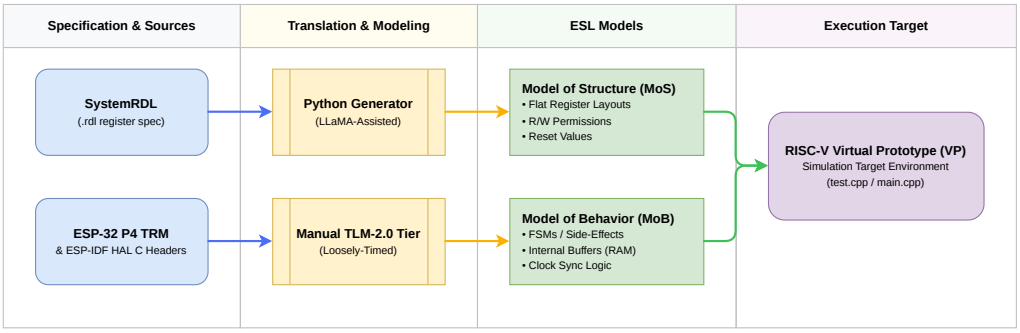


Figure 4.7: Bipartite ESL Modeling Flow: Automated Model of Structure (MoS) Generation via LLM and SystemRDL combined with Manual Model of Behavior (MoB) for complete peripheral validation.

Software Test Bench Framework

To validate the structural integrity of generated hardware structures from a software perspective, a standalone bare-metal C application (`main.c`) was compiled into ELF binary format using the cross-compiler toolchain. This test bench bypasses operating system abstraction layers, issuing volatile memory pointer operations directly targeting hardware base configurations.

The test application targets specific physical base addresses from the ESP32-P4 memory map, such as UART0 at `0x500CA000`. The validation software executes a sequential loop writing character sequences directly into `UART_FIFO_REG`, evaluating the core's capability to process back-to-back register transactions.

Structural Generation Results

The automated Python-based synthesis framework successfully generated complete register definitions for each target peripheral: 37 registers for UART, 35 for I2C, and 18 unique functional register types for LEDC.

The firmware test bench was executed within the top-level test environment (`test.cpp`). Standard console output captured during this automated structural testing phase is shown below:

```
--- Starting the peripheral test ---
Testing UART0 register retention SUCCESS
Testing UART0 Reset Value SUCCESS
Testing UART0 Read-Only protection SUCCESS
Testing I2C0 register retention SUCCESS
Testing LEDC register retention SUCCESS
--- Test finished ---
```

This output confirms successful initialization of the interrupt architecture and structural layout integrity:

1. **Interrupt Architecture Setup:** The platform correctly establishes the Core Local Interrupt Controller (CLIC) environment, generating 32 independent interrupt request (IRQ) channels connected to hardware thread 0 (`hart 0`).
2. **Register Retention Checks:** The SUCCESS status for UART0, I2C0, and LEDC register retention verifies that underlying memory-backed storage cells preserve written state values across independent simulation cycles without leakage or corruption.
3. **Reset Value Validation:** The UART0 reset value check confirms that synthesized C++ structural fields match the default physical silicon states mandated by the hardware specification at system boot.
4. **Boundary Enforcement:** The UART0 Read-Only protection mechanism confirms that auto-generated masking logic successfully drops illegal store instructions targeting status fields, preventing unauthorized software interactions from compromising critical internal states.

Functional testing confirms that this automated Model of Structure (MoS) achieved complete structural precision. The auto-generated layers successfully handled basic read/write access boundaries, maintained expected reset values, and enforced bit retention across successive simulation cycles without layout mismatches.

4.4 Phase 2: Behavioral Evaluation of Manually Modeled Peripherals

While automated generation verified structural register layouts, flat register maps lack the complex internal logic and finite state machines necessary to execute production-grade software stacks. To achieve deep functional fidelity, Phase 2 implements manual modeling at the loosely-timed (LT) abstraction tier in accordance with the IEEE 1666-2023 SystemC standard [20], resulting in development of `my_uart.h`, `my_i2c.h`, and `my_ledc.h`. This approach aligns with systematic studies [3] showing that loosely-timed behavioral models dominate verification of functional capabilities and system integration interfaces.

Design Basis and Ground Truth Alignment

The manual peripheral models utilize a dual-source validation strategy. Structural ground truth derives directly from the detailed register specifications of the *ESP32-P4 Technical Reference Manual (TRM)* [5]. Behavioral ground truth derives from official low-level driver source code within the Espressif IoT Development Framework (ESP-IDF) [42], including `uart.h`, `i2c.h`, and `ledc.h`.

To bridge abstract memory layouts and real hardware behavior, several high-fidelity architectural adjustments were manually integrated into the SystemC peripheral models.

High-Fidelity Internal Memory Mapping

Production software drivers frequently perform block data transfers relying on internal peripheral memory arrays. To accommodate these operations, dedicated internal RAM buffers were integrated into manual transactional handling logic. A 32-byte internal random-access memory array was mapped inside the I2C module at register offset `0x0100`,

supporting the official driver function `i2c_ll_write_tx_by_nonfifo` which directly targets this register space.

Similarly, a specialized SRAM buffer for LED PWM gamma-correction lookup values was integrated within the LEDC module at register offset `0x0800`. Without authentic internal storage arrays, such firmware operations trigger immediate bus faults or store corrupted payloads.

Clock-Domain Synchronization Logic

Physical silicon architectures implement distinct, asynchronous clock domains to isolate high-frequency processor cores from low-frequency peripheral buses. When software modifies configuration registers, these changes must be synchronized via internal hardware handshakes. The manual models simulate these mechanisms by introducing specialized state-tracking logic for synchronization flags, including the `reg_update` bit in the UART model and the `conf_update` bit in the I2C module.

To maintain strict evaluation consistency across both modeling paradigms, a secondary simulation platform top-level wrapper (`main.cpp`) was constructed to link manual peripherals together. This platform executes the exact same cross-compiled ELF application binary derived from the software test bench, ensuring complete testing uniformity.

The overall Phase 2 behavioural evaluation workflow is illustrated in Figure 4.8.

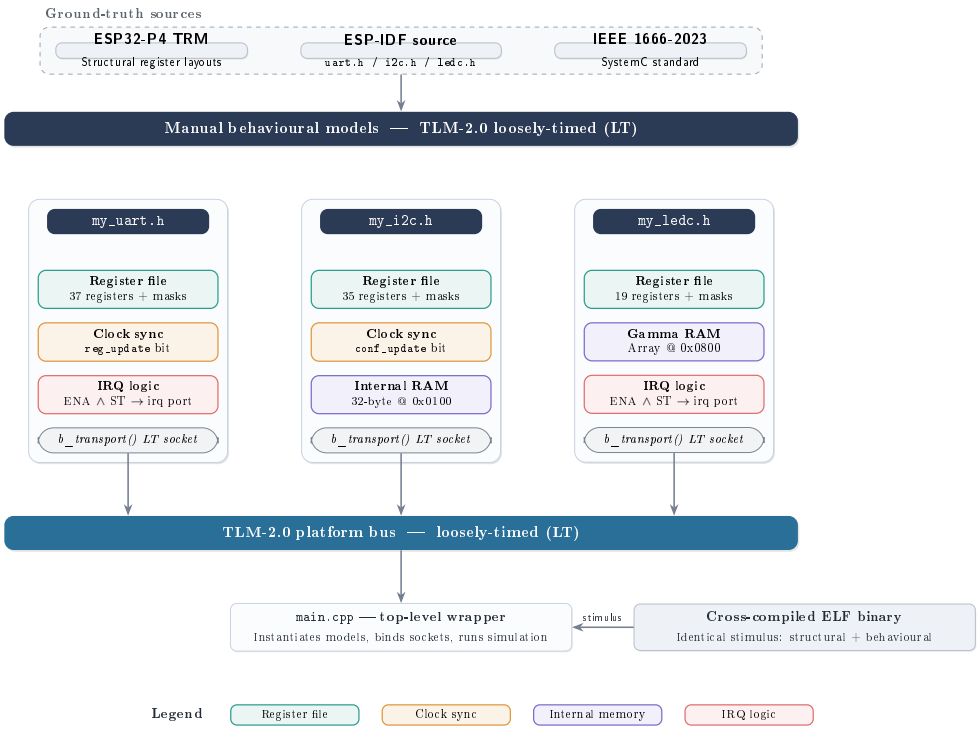


Figure 4.8: Phase 2 behavioural evaluation flow: hand-written UART, I2C, and LEDC TLM models integrated into the RISC-V virtual platform and validated using the same bare-metal firmware as Phase 1.

Testing Against Production Drivers: The UART Hang Problem

While Phase 1 succeeds with synthetic test firmware (`main.c`) that treats registers as simple storage cells, real embedded drivers rely on implicit hardware behaviors that Phase 1 cannot model. These include synchronization bits that self-clear after internal handshakes, interrupt thresholds that fire at configurable watermarks, and FIFO buffering where writes trigger side-effects.

Initial Discovery: Firmware Stall

During early execution testing of the behavioral models against production Espressif ESP-IDF drivers, the host system terminal successfully displayed initial peripheral verification blocks but consistently stalled midway through the final character confirmation string. The execution abruptly truncated after processing the first eight bytes of the test-completion message:

```
--- Starting the peripheral test ---
Testing UART0 register retention SUCCESS
Testing UART0 Reset Value SUCCESS
Testing UART0 Read-Only protection SUCCESS
Testing I2C0 register retention SUCCESS
Testing LEDC register retention SUCCESS
--- Tes
```

Root Cause: Missing Synchronization Handshakes

To determine the root cause of this simulation freeze, deep runtime analysis was conducted. In production hardware environments, low-level drivers execute a blocking `while` loop immediately after configuring a device. The software asserts a synchronization bit—such as `UART_REG_UPDATE_REG` or I2C `conf_update` flag—and spins until internal hardware handshakes complete, indicated by the silicon automatically clearing the tracking bit back to zero.

During Phase 1 testing with flat register maps, the model functioned as a basic storage cell. When software wrote 1 to the synchronization register, the model retained that value indefinitely. Consequently, when executed against authentic drivers, firmware became trapped in an infinite polling loop, stalling because the register never dropped back to zero.

Secondary Issue: TX FIFO Threshold Interrupt

Even after correcting the register update handshake to force synchronization bits back to zero, firmware still experienced a hard stall at exactly the eighth byte (`--- Tes`). Investigating the official ESP-IDF UART driver revealed a secondary constraint: the driver relies on the hardware's **TX FIFO Empty Interrupt Threshold** (`tx_mem_cnt_thr`).

Rather than waiting for the FIFO queue to deplete entirely to zero, the driver sets an interrupt watermark (typically 8 bytes remaining). Because the initial transaction-level model only asserted the TX interrupt flag when the internal FIFO became completely empty, the driver was starved of its expected threshold event. This mismatch trapped the processor execution in an unrecoverable driver status-polling spin.

Resolution: Modeling Threshold-Based Interrupts

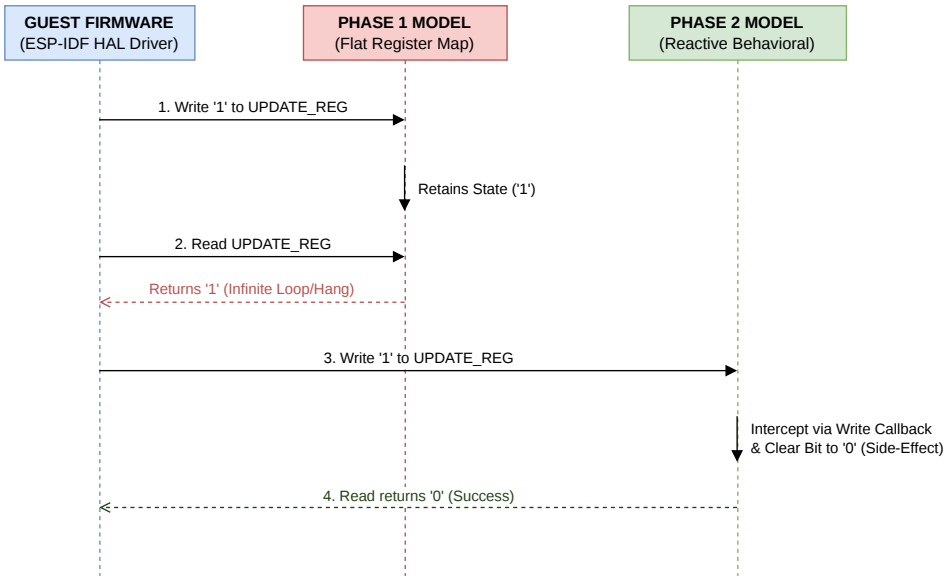


Figure 4.9: Comparison of Guest Firmware Interactivity: Static Register Map (Phase 1) vs. Reactive Behavioral Model (Phase 2) with Write Callbacks to Prevent Driver Deadlocks.

To resolve both synchronization deadlock and threshold starvation, the manual UART model was updated to simulate hardware-accurate interrupt behavior:

1. Synchronization bits (`UART_REG_UPDATE_REG`, `I2C_CONF_REG`) are now forced back to zero immediately within the write callback, simulating zero-delay hardware handshakes across clock domains.
2. The TX FIFO interrupt is asserted proactively as bytes drain from the queue, not only when completely empty. This ensures the driver's threshold-based polling loop receives expected interrupt signals at configurable watermarks.

With these behavioral refinements, the firmware successfully bypassed both deadlock scenarios and executed to completion without modification.

Verification: Successful Execution

Following these behavioral adjustments, the platform was re-executed and successfully bypassed both synchronization and threshold deadlocks:

```
--- Starting the peripheral test ---
Testing UART0 register retention SUCCESS
Testing UART0 Reset Value SUCCESS
Testing UART0 Read-Only protection SUCCESS
Testing I2C0 register retention SUCCESS
Testing LEDC register retention SUCCESS
--- Test finished ---
```

This demonstrates that for a loosely-timed virtual platform to be truly "driver-compatible," it must move beyond basic register address mapping to accurately model

internal system synchronization mechanics and interrupt-driven control flows.

GDB Diagnostic Stack Analysis

To mathematically prove that the modified simulation environment executed correctly rather than hanging on invalid bus access or infinite synchronization loops, a remote diagnostic session was established by binding GNU Debugger (GDB) to the active SystemC simulator process. Issuing manual interrupt (`SIGINT`) trapped the execution context during active cycle execution:

```
(gdb) backtrace
#0  0x00007ffff7b3bc60 in free@plt () from /lib/.../libstdc++.so.6
#1  0x000055555575189a in CombinedMemoryInterfaceBase<...>::_do_transaction
    at /root/BuddelChip/src/riscv-vp/iss/common/mem.h:109
#2  0x000055555574b6ac in CombinedMemoryInterfaceBase<...>::_raw_load_data
    at /root/BuddelChip/src/riscv-vp/iss/common/mem.h:124
#3  0x0000555555741704 in CombinedMemoryInterfaceBase<...>::load_instr
    at /root/BuddelChip/src/riscv-vp/iss/common/mem.h:217
#4  0x000055555578b946 in rv32::ISS::exec_step
    at /root/BuddelChip/src/riscv-vp/iss/rv32/iss.cpp:741
```

Inspecting active frame variables within the core Instruction Set Simulator exposed the exact program counter state:

```
(gdb) frame 4
(gdb) print this->pc
$1 = 1341129364
```

Converting the base-10 address to hexadecimal:

$$1341129364 = 0x4FF00094 \tag{4.1}$$

By checking the platform's layout configuration (`linker.ld`), the High-Performance Layer-2 Memory (`HP_L2MEM`) segment boundaries are:

```
MEMORY {
    ram (rwx) : ORIGIN = 0x4FF00000, LENGTH = 0x100000
}
```

Calculating the instruction pointer offset relative to memory base:

$$\Delta_{\text{offset}} = 0x4FF00094 - 0x4FF00000 = 0x94 \quad (148 \text{ bytes}) \tag{4.2}$$

This diagnostic evaluation provides mathematical proof that the simulator executed safely inside the valid `.text` section. The 148-byte offset corresponds precisely to the compiled assembly of the final safety spin lock (`while(1)`), confirming that the application successfully passed all peripheral validation blocks, completed its transmission sequences, and reached terminal state exactly as intended.

4.5 Comparison of Phase 1 and Phase 2 Approaches

This section compares the automatically generated models from Phase 1 with the manually refined behavioral models from Phase 2, highlighting their respective contributions to structural accuracy and functional realism.

Baseline Equivalency and Structural Preservation

The empirical evaluation of the virtual prototype reveals a fundamental distinction between automated structural generation (Phase 1) and manual behavioral modeling (Phase 2). The auto-generated peripherals, compiled from machine-readable SystemRDL descriptions using the LLaMA 3.3 70B Large Language Model, demonstrated exceptional structural precision. The synthesis pipeline accurately identified and mapped all 37 registers for UART, 35 for I2C, and 18 unique functional register types for LEDC. This establishes a robust "Model of Structure" (MoS), which excels at validating memory map alignment, confirming boot-up reset values, and enforcing hardware read/write permission boundaries early in the engineering lifecycle.

Model of Structure (MoS) vs Model of Behavior (MoB)

The evaluation distinguishes two orthogonal aspects of peripheral modeling:

Model of Structure (MoS) captures the static register layout: offsets, field widths, reset values, read/write permissions, and access masks. This is the domain of automated LLM-assisted generation from SystemRDL.

Model of Behavior (MoB) captures the dynamic response to register writes: synchronization bit auto-clearing, interrupt threshold firing, internal state transitions, and side-effects on subsequent reads. This requires manual modeling guided by hardware specifications and driver source code.

When subjected to the bare-metal verification firmware (`main.c`), both Phase 1 auto-generated and Phase 2 manually-refined models yielded identical successful console outputs. This establishes **Baseline Equivalency**, confirming that manual integration of complex transactional logic did not corrupt the underlying structural integrity or memory offsets established during automated generation.

The Necessity of the Behavioral Model

While flat auto-generated register maps excel at structural validation (MoS), they expose critical behavioral deficiencies when scaled to production environments. Auto-generated models function strictly as passive data storage cells. When evaluated against authentic, production-grade firmware drivers (Espressif ESP-IDF HAL), Phase 1 simulation becomes trapped in infinite polling loops.

This deadlock occurs because flat register models cannot simulate:

- Self-clearing behavior of hardware synchronization bits (`UART_REG_UPDATE_REG`, `I2C_CONF_REG` flags)
- Interrupt thresholds that fire at configurable watermarks (`tx_mem_cnt_thr`)
- FIFO buffering where writes trigger reactive state changes
- Status flags that change autonomously based on internal peripheral state

In physical silicon, asserting configuration bits triggers internal hardware handshakes across clock domains, after which control logic automatically clears the tracking bit. In Phase 2, this reactive control loop was successfully integrated by overriding TLM 2.0 write callback paths. The manual transaction handler intercepts incoming store payloads and triggers immediate, state-dependent transaction side-effects: forcing target bits back to

zero within a delta-cycle timeline and proactively asserting interrupt thresholds as buffers drain.

By modeling these reactive control dynamics, Phase 2 bridges the critical gap between abstract memory layouts and authentic hardware behavior. It ensures the virtual platform retains structural precision while achieving execution fidelity necessary to run unmodified, production-grade OS and driver stacks natively without simulation stalls [1].

4.6 Comparison of the Two Approaches

The evaluation exposes a clear division of labour between automated structural generation (Phase 1) and manual behavioural modeling (Phase 2). The auto-generated peripherals, compiled from machine-readable SystemRDL with the assistance of the LLaMA 3.3 70B language model, were structurally excellent: the pipeline identified and mapped all 37 UART registers, all 35 I2C registers, and all 18 LEDC functional register types. This **Model of Structure (MoS)** is ideal for validating memory-map alignment, confirming reset values, and enforcing access permissions early in the lifecycle.

What it cannot supply is behaviour. As passive storage cells, the generated maps drove authentic ESP-IDF drivers into infinite polling loops, because a flat register cannot reproduce the self-clearing of a hardware synchronisation bit such as `UART_REG_UPDATE_REG` or the I2C `I2C_CONF_REG` flag. In silicon, asserting such a bit launches a cross-domain handshake after which the control logic clears it automatically. The manual **Model of Behavior (MoB)**—`my_uart.h`, `my_i2c.h`, and `my_ledc.h`—restores this by overriding the TLM-2.0 write callbacks: the handler intercepts stores to synchronisation offsets and forces the target bit back to zero within a delta cycle. Modeling these reactive dynamics is precisely what lets firmware run to completion without stalls [1].

4.7 Productivity and Scalability

The contrast between the phases translates into a substantial productivity gain. Transcribing a peripheral's register map from a reference manual into C++ or SystemC boilerplate by hand is slow and error-prone, and is realistically a full engineer-day per block once the bit-masking and offset bookkeeping are accounted for—a cost confirmed directly by the CLIC and ISP of Section 4.2, each of which consumed several days. The automated pipeline reduced this to a few minutes of runtime followed by roughly fifteen minutes of audit to check parametric corner cases or reconcile cross-page reset values.

At the scale of modern SoCs—interconnects that routinely expose more than 25,000 on-chip registers [43]—manual transcription is not merely slow but commercially unviable and dangerous: a single mistyped offset or array index can plant a runtime bug that is extremely hard to trace. Treating SystemRDL as a *single source of truth* keeps the architectural specification and the executable SystemC in lock-step and eliminates specification drift.

The generator also absorbs differences in address spacing through parametric truncation. Rather than reserving rigid, oversized blocks, it derives a bitwise mask from each

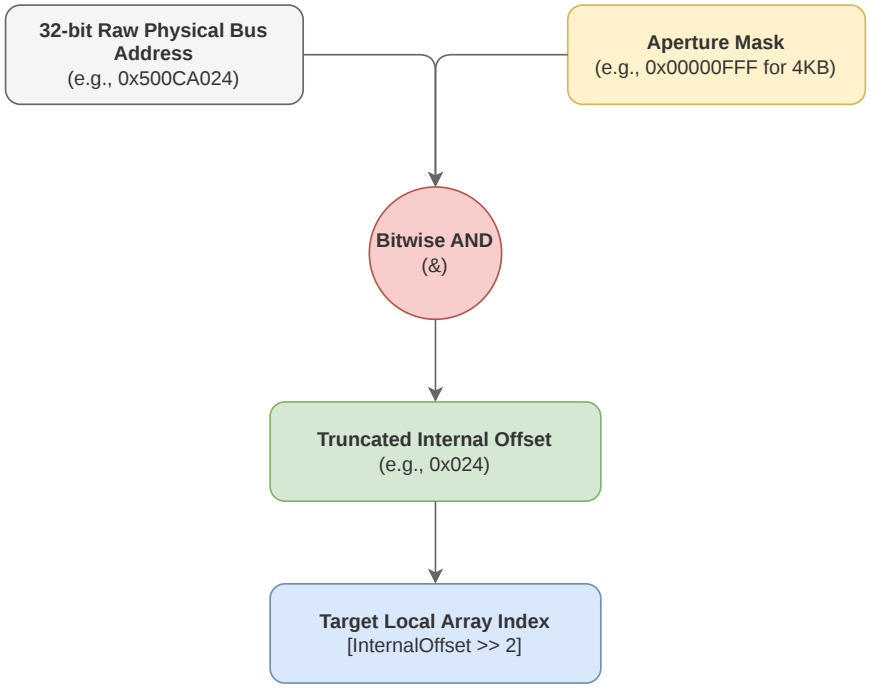


Figure 4.10: Parametric aperture masking and bitwise offset truncation for mapping local storage-array indices.

block's aperture size,

$$\text{Mask} = \text{ApertureSize} - 1, \tag{4.3}$$

and slices incoming 32-bit bus addresses to internal indices with a single operation,

$$\text{InternalOffset} = \text{BusAddress} \,\&\, \text{Mask}. \tag{4.4}$$

The same mechanism scales without modification from a 4 KB peripheral window to a 256 KB USB-OTG aperture.

4.8 Generalisation and Reusability

The framework is built for reuse, achieved by keeping three design concerns—communication interface, internal behaviour, and structural data—in separate, independently maintainable layers, in the spirit of the classic orthogonalisation of concerns [44].

Structural reuse Capturing the register maps in standard SystemRDL [43] makes the `.rdl` files independent of the simulation target, so they can feed downstream EDA flows to produce documentation, synthesisable RTL, or UVM verification blocks.

Infrastructure reuse The generic `peripheral_template.hpp` wrapper provides a TLM-2.0 blocking-transport interface that handles bus communication independently of the peripheral type, parameterised through C++ templates:

```
template <typename RegMapType, unsigned int APERTURE_SIZE>
class PeripheralTemplate : public sc_core::sc_module {
public:
    tlm_utils::simple_target_socket<PeripheralTemplate> socket;
    RegMapType registers;
    // Generic b_transport implementation...
};
```

Listing 4.11: SystemC Core Write Callback Handler for UART Peripheral.

The Python generator injects each peripheral's register layout and interrupt channels into this shell, so the identical infrastructure serves the UART, I2C, and LEDC modules alike.

Driver compatibility Aligning the internal memory regions and behavioural side-effects with the official Espressif HAL headers lets the manual models act as high-fidelity golden references, which can be dropped into separate test benches or regression suites to validate production binaries without touching the guest code.

4.9 General Discussion

– The initial execution "hang" observed in the console output highlighted an important challenge in virtual-prototype development: a platform becomes truly driver-

compatible only when it accurately models internal synchronization mechanisms, not merely register maps.

- The LLM-based extraction approach achieved excellent results for structural modeling, reaching **99.6% structural precision** on the UART peripheral. This confirms the effectiveness of automated generation for extracting register layouts and hardware structure.
- However, a clear *semantic boundary* was identified when generating behavioral models. As previously observed during the CLIC and ISP modeling phases (Section 4.2), current language models struggle to infer:
  - Complex timing dependencies,
  - Multi-register state transitions,
  - Interrupt hierarchies,
  - Architectural relationships distributed across multiple chapters of a hardware manual.
- This limitation reappeared during validation of the ISP behavioral model (Section 4.2). Although the model successfully passed the synthetic-gradient testbench, execution of the production firmware `isp_multi_pipeline.elf` resulted in an immediate software stall.
- Instruction-trace analysis revealed that the ISP driver relies on proprietary vendor-specific instruction extensions that are not documented in the public ESP32-P4 reference manual.
- When these custom instructions were encountered:
  - The baseline ISS could not decode the opcodes.
  - An illegal-instruction exception was raised.
  - The processor set `mcause = 2`.
  - The firmware entered an infinite trap-handling loop during early boot.
- This result demonstrates that undocumented ISA extensions represent a fundamental limitation for standard virtual platforms unless matching instruction decoders are implemented alongside the simulation model.
- To ensure reproducible and fully documented experiments, the quantitative evaluation was restricted to the UART, I2C, and LEDC peripherals, whose register interfaces and instruction requirements are fully described in public documentation.
- The proposed **bipartite methodology** combines:
  - Automated LLM-based generation for structural boilerplate.
  - Manual engineering refinement for behavioral modeling.
- This shift-left strategy significantly reduces development effort while allowing engineers to focus on architectural validation, behavioral correctness, and early software integration instead of manual register transcription.
- Future work should investigate multi-modal hardware-document understanding, combining textual descriptions, register tables, and architectural diagrams to automatically infer behavioral logic in addition to structural information.

4.10 Conclusion

The evaluation confirmed that automated generation is highly effective for structural peripheral modeling, whereas manual refinement is required to capture dynamic hardware behavior. By combining both approaches, the proposed methodology achieves accurate, reusable, and firmware-compatible virtual prototypes while significantly reducing development effort.

5

Synthesis and Future Work

This final chapter synthesizes the core findings of our research into automated hardware modeling and outlines strategic pathways for future extension.

We review the automated pipeline's performance in bridging the gap between natural language manuals and functional transaction-level virtual platforms.

Following this synthesis, we discuss open challenges and promising avenues to further mature this methodology for complex industrial system-on-chip architectures.

5.1 Thesis Summary and Key Achievements

This thesis has presented, implemented, and verified a modular, end-to-end automated generation toolchain designed to alleviate a critical bottleneck in Electronic System-Level (ESL) design: the manual, error-prone transcription of virtual prototype peripheral models from natural-language documentation. By leveraging Large Language Models (LLMs) as intelligent semantic parsers, the developed framework successfully bridges the gap between unstructured Technical Reference Manuals (TRMs) and executable, transaction-level simulation environments [45].

The structural core of this research is a single-entry Python processing pipeline (`run_pipeline.py`) that orchestrates six independent, decoupled stages: automated PDF preprocessing, layout extraction, boundary-aware text chunking, formal register map description generation via the SystemRDL standard [6], compiler-driven verification with iterative repair, and template-guided SystemC/TLM-2.0 code generation [29]. This modularity ensures that individual components—such as the generative AI backend, the verification compiler, or the target template engines—can be seamlessly updated or replaced without disrupting the overarching architectural workflow.

A foundational pillar of this methodology was the strategic utilization of the open-weight LLaMA 3.3 70B model deployed on the high-throughput Groq inference platform. By fixing the decoding temperature to a strict value of `0`, the model's probabilistic nature was regularized to enforce deterministic token selection. This constraint is vital when synthesizing hardware specifications, where even a single bitwise deviation introduces catastrophic runtime software flaws. By pairing this deterministic model configuration with an automated compiler feedback loop driven by the `peakrdl` compiler framework,

the pipeline achieved a standalone SystemRDL compilation success rate of 99% after minimal refinement cycles, successfully recovering from syntax omissions without manual developer intervention.

5.2 Synthesis of Findings across Evaluated Peripherals

The evaluation phase detailed in Chapter 4 demonstrated both the structural precision of the extraction pipeline and its practical utility when integrating auto-generated blocks into a system-level virtual platform.

Peripheral Extraction and Generation Metrics

The automated framework was empirically evaluated by targeting three critical on-chip peripherals from Espressif's advanced ESP32-P4 system-on-chip (SoC) architecture [5], yielding substantial structural validation metrics:

- **UART Peripheral:** The toolchain achieved a structural accuracy score of 99.6%, correctly extracting and mapping all 37 distinct configuration and status registers. Minor omissions were strictly confined to clock-dependent parametric reset expressions and cross-page footnote entries.
- **I2C Peripheral:** All 35 target memory-mapped registers were successfully instantiated within the intermediate representation, yielding a field-name accuracy of 98.3% and an address offset precision of 97.1%. The cumulative accuracy score of 96.6% was primarily bounded by a drop in reset-value accuracy to 92.4%, caused by the model's intrinsic inability to resolve complex hardware timing constants that were documented outside the isolated register tables in separate protocol timing diagrams.
- **LEDC Peripheral:** The pipeline demonstrated exceptional robustness when confronted with dense, repetitive hardware structures. It successfully parsed the Pulse Width Modulation (PWM) controller's arrayed channel and timer topologies, accurately interpreting parametric stride offsets expressed via algorithmic definitions such as `0x00A0 + 0x14 × n`.

Across all tested components, the pipeline compressed the engineering modeling lifecycle from an estimated full working day per peripheral down to a few minutes of automated computational execution, proving that executing open-weight models on modern inference tiers can drastically accelerate production timelines at negligible operational cost.

Virtual Platform Integration and Bug Resolution

During the system-level integration phase within the target RISC-V virtual platform, several complex hardware-software interaction anomalies were identified and resolved, ensuring seamless compatibility between the auto-generated peripheral blocks and the compiled bare-metal target software:

- **Word-Alignment Indexing Errors:** A severe address indexing vulnerability was eliminated within the transaction handling loop (`b_transport()`). Raw inbound byte addresses applied directly as local array indices provoked out-of-bounds host memory corruption during multi-byte transfers. Implementing a strict bitwise right-shift operation (`addr >> 2`) mathematically normalized incoming byte addresses into aligned 32-bit word indices, enforcing absolute memory safety and silicon-accurate register targeting under heavy software workloads.
- **Custom Framework Interrupt Binding:** Because the virtual platform's Core Local Interrupt Controller (CLIC) module uses an optimized, inheritance-based `interrupt_gateway` model, standard SystemC signal ports (`sc_signal`) could not be bound natively. To circumvent this, the Jinja2 template engine was extended to inject an inline pointer link and a dedicated `setup_interrupts` routine. This allowed the synthesized peripherals to reference the system CLIC directly, tripping virtual interrupts via fast C++ function calls with minimal simulation overhead.
- **Interconnect Alignment and Assertions:** Catastrophic Load Page Fault exceptions (`mcause: 0xd`) and structural address collisions were resolved by fully refactoring the system bus topology within `test.cpp`. The unified Interrupt Matrix was relocated to its true physical base address at `0x500D6000`, isolating it from the CLIC memory layout. Concurrently, expanding the `UART0` memory aperture to its native 4 KB window (up to `0x500CAFFF`) enabled bare-metal test code to poll extended status fields safely. Finally, shifting the validation firmware verification checks to utilize the generated write-masks guaranteed that the validation code strictly respected hardware-enforced reserved bit boundaries rather than hitting empty FIFO holes.

Bipartite Strategy: Optimal Path for Modern SoC Design

The preceding results demonstrate that the optimal strategy for industrial virtual platform development is *bipartite*: automated structural generation for register maps, addressing, and hardware topology, combined with targeted manual refinement for behavioral semantics and complex algorithmic subsystems. This thesis validates that LLM-driven tools excel at extraction and synthesis of well-defined specification layers—where rules are explicit and deterministic—but require human expertise for interpreting informal prose, cross-document context, and FSM-level behavior. By clearly delineating this boundary, development teams can allocate engineering effort efficiently: invest automation infrastructure to eliminate tedious register table transcription, and reserve skilled engineers for the irreducibly creative work of functional validation and complex control logic design.

5.3 Limitations and Open Challenges

While the automated generation toolchain represents a major advancement in accelerating virtual prototype synthesis, several intrinsic limitations remain open challenges:

- **Semantic Boundary Bottleneck:** The primary limitation stems from the fact that a text-based LLM inspecting isolated register tables lacks global architectural context. When initialization parameters rely on clock-gating variables, footnotes, or finite-

state machine (FSM) diagrams scattered across distant chapters of a TRM, the model frequently defaults to zero or omits crucial register properties.
- **Context Window Fragmentation:** Large peripheral specifications containing hundreds of registers inevitably split across multiple document processing chunks. This fragmentation breaks the sequential continuity of register map layouts, leading to redundant prompt insertion overhead and occasional coordinate misalignment of extended bitfields.

5.4 Directions for Future Work

To build upon the contributions established in this thesis and further advance the state of the art in automated electronic design automation (EDA), several promising research avenues are proposed:

- **Multi-Modal and Context-Aware Ingestion:** Future iterations of the pipeline should incorporate multi-modal vision-language models capable of concurrently processing unstructured prose alongside hardware block diagrams, address map tables, and timing waveforms [46]. This extension will directly address the documented failure modes in this work: cross-page footnote resolution and extraction of protocol timing constants from waveform diagrams. Specifically, the I2C reset-value accuracy gap (92.4%) and UART cross-page omissions (footnote annotations) can be recovered by allowing the model to visually ground register descriptions in their corresponding architectural diagrams and timing specifications. This multi-modal strategy will enable synthesis of the global register initialization context that pure text extraction cannot access.
- **Local Fine-Tuning on Domain-Specific EDA Datasets:** Fine-tuning smaller, open-weight models (e.g., 8B or 14B parameter architectures) directly on verified SystemRDL codebases [29], Accellera specifications, and SystemC/TLM design patterns could yield comparable or superior structural generation quality locally. This strategy eliminates cloud-tier API dependency and safeguards proprietary corporate intellectual property.
- **Automated Functional Behavior Synthesis:** The current framework excels at generating accurate structural wrappers, register blocks, and hardware bitmasks. Extending the prompt-engineering framework to ingest functional prose from datasheets and synthesize algorithmic C++ execution bodies directly within the transaction-handling loop represents the next logical milestone in complete virtual platform automation.
- **Advanced Multi-Peripheral Arbitration and Stress Testing:** Expanding the verification environment to support automated, synthetic software driver generation would enable extensive system stress testing. Introducing randomized bus contention loops within the `SimpleBus` interconnect would provide a vehicle to validate the robustness of the parametric address masking layers under highly saturated, concurrent multi-core processing profiles.

5.5 Conclusion

This chapter confirmed that the automated pipeline achieves high extraction accuracy while greatly reducing modeling time, validating a bipartite approach where automation handles structural extraction and humans handle behavioral refinement. Identified limitations point toward multi-modal ingestion and fine-tuning as next steps.

Bibliography

[1] Ghenassia, F. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer Science & Business Media, 2006.

[2] Grötker, T., Liao, S., Martin, G., and Swan, S. *System Design with SystemC*. Springer, 2002.

[3] Mahmoudi, A., Nešković, A., Thermann, C., Sehm, R., Hübner, C., Plattenteich, T., Meyer, R., Buchty, R., Berekovic, M., and Mulhem, S. A Systematic Mapping Study on SystemC/TLM Modeling Capabilities in New Research Domains. In: *ACM Transactions on Design Automation of Electronic Systems* 30(4):1–41, 2025. DOI: 10.1145/3735641.

[4] Zeng, H. et al. Symbolic execution for software-based interrupt controllers. In: *Proceedings of the Design, Automation & Test in Europe (DATE) Conference*. IEEE. 2014, pp. 1–6.

[5] Espressif Systems *ESP32-P4 Technical Reference Manual*. Version 1.1. Espressif Systems. 2024.

[6] *SystemRDL 2.0 Register Description Language Specification*. Accellera Systems Initiative. 2018.

[7] *Magillem Registers: Hardware/Software Interface Automation Framework*. Technical Whitepaper. Arteris Inc. 2025.

[8] Meyer, R., Farkas, B., Shah, S. A. A., and Berekovic, M. Transparent SystemC Model Factory for Scripting Languages. In: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVCon Europe)*. 2022.

[9] Tabacaru, L., Meyer, R., and Berekovic, M. Automated Verification in Electronic Co-Design: Shift-Left Strategies for Industrial SoC Design. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2026.

[10] Synopsys Inc. *Virtual Prototyping for Hardware/Software Co-Verification*. Technical Whitepaper. Synopsys Inc., 2024.

[11] *RISC-V Virtual Platform*. Open-source SystemC-based virtual platform for RISC-V ISA simulation. 2024.

[12] Liu, S., Xiao, W., Li, Y., Pan, D. Z., and Hu, Z. RTLCoder: Fully Open-Source and Efficient LLM-Assisted RTL Code Generation Technique. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[13] Liu, T., Tian, Q., Ye, J., Fu, L., Su, S., Li, J., Wan, G.-W., Zhang, L., Wong, S.-Z., Wang, X., et al. ChatChisel: Enabling Agile Hardware Design with Large Language Models. In: *Proceedings of the 2nd International Symposium of Electronics Design Automation (ISEDA)*. IEEE. 2024, pp. 710–716.

[14] Niu, J., Liu, X., Niu, D., Wang, X., Jiang, Z., and Guan, N. ReChisel: Effective Automatic Chisel Code Generation by LLM with Reflection. In: *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2025, pp. 1–7.

[15] Thakur, S., Ahmad, B., Fan, Z., Pearce, H., Tan, B., Karri, R., Dolan-Gavitt, B., and Garg, S. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In: *arXiv preprint arXiv:2212.11140*, 2022.

[16] Wan, G.-W., Wang, Y., Wong, S.-Z., Zhang, J., Xing, M., Jiang, Z., Guan, N., Wang, Y., Xu, N., Xu, Q., et al. *GenBen: A Generative Benchmark for LLM-Aided Design*. 2025.

[17] Ye, J., Liu, T., Tian, Q., Su, S., Jiang, Z., and Wang, X. *ChatModel: Automating Reference Model Design and Verification with LLMs*. 2025.

[18] Li, R., Xiong, J., and Wang, X. *iDSE: Navigating Design Space Exploration in High-Level Synthesis using LLMs*. 2025.

[19] Abdollahi, M., Yeganli, S. F., Baharloo, M., and Baniasadi, A. Hardware Design and Verification with Large Language Models: A Scoping Review, Challenges, and Open Issues. In: *Electronics* 14(1):24, 2025.

[20] IEEE Computer Society *IEEE Std 1666-2023: IEEE Standard for SystemC Language Reference Manual*. IEEE Standards Association. New York, NY, USA, 2023.

[21] Braun, G., Leupers, R., and Meyr, H. Rapid Prototyping Methodologies for Hardware-Software Co-Verification. In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 1–6.

[22] *Machineware VCML: Virtual Platform Framework*. SystemC/TLM virtual platform library for SoC simulation. 2024.

[23] Xia, C. S., Wei, Y., and Zhang, L. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2024, pp. 1–14.

[24] Wan, G.-W., Su, S., Zhang, J., Wong, S.-Z., Xing, M., Ji, L., Jiang, Z., Wang, X., and Yang, J. ChatTest: Coverage-Enhanced Testbench Generation for Agile Hardware Verification with LLMs. In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE/ACM. 2026, pp. 1–6.

[25] Zhang, Y., Ye, J., Yang, J., and Wang, X. RefEvo: Agentic Design with Co-Evolutionary Verification for Agile Reference Model Generation. In: *arXiv preprint arXiv:2604.24218*, 2026.

[26] Wang, Y., Wu, Z., Yao, J., and Su, M. TDAG: A Multi-Agent Framework Based on Dynamic Task Decomposition and Agent Generation. In: *Neural Networks* 185, 2025.

[27] Xing, C., Wong, S.-Z., Wan, X., Lu, Y., Zhang, M., Ma, Z., Qi, L., Li, Z., Guan, N., Jiang, Z., et al. *ChipMind: Retrieval-Augmented Reasoning for Long-Context Circuit Design Specifications*. 2026.

[28] Densmore, D., Passerone, R., and Sangiovanni-Vincentelli, A. A Taxonomy of Abstraction Levels for Electronic System Design. In: *Methodologies and Technologies for Networked Enterprises*. Springer, 2006, pp. 263–282.

[29] *Transaction Level Modeling (TLM) 2.0 Language Reference Manual*. Open SystemC Initiative (OSCI), now Accellera Systems Initiative. 2009.

[30] Lalonde, A. *systemrdl-compiler: Parser and elaborator for the SystemRDL 2.0 register description language*. https://github.com/SystemRDL/systemrdl-compiler. 2024.

[31] Artifex Software *PyMuPDF: Python bindings for MuPDF*. https://pymupdf.readthedocs.io. 2024.

[32] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is All You Need. In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.

[33] Grattafiori, A. et al. The Llama 3 Herd of Models. In: *arXiv preprint arXiv:2407.21783*, 2024.

[34] OpenAI GPT-4o System Card. In: *arXiv preprint arXiv:2410.21276*, 2024.

[35] Jiang, A. Q. et al. Mistral 7B. In: *arXiv preprint arXiv:2310.06825*, 2023.

[36] Yang, A. et al. Qwen2 Technical Report. In: *arXiv preprint arXiv:2407.10671*, 2024.

[37] Zhang, P., Zeng, G., Wang, T., and Lu, W. TinyLlama: An Open-Source Small Language Model. In: *arXiv preprint arXiv:2401.02385*, 2024.

[38] Abts, D., Ross, J., Sparling, J., Wong, M., Lawrence, M., Lo, G., Miller, P., Talpes, E., Bannon, J., Shufstein, D., et al. Think Fast: A Low Latency, High Throughput Accelerator for Deep Learning. In: *arXiv preprint arXiv:2008.06754*, 2020.

[39] Pallets Projects *Jinja Documentation (Version 3.1.x)*. https://jinja.palletsprojects.com/. 2024.

[40] Brank, J. et al. Challenges in virtual prototyping for modern SoCs. In: *Proceedings of the Design, Automation & Test in Europe (DATE) Conference*. 2022.

[41] Modin, A. et al. Comparative methodologies in ISA simulation research. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[42] Espressif Systems *Espressif IoT Development Framework (ESP-IDF) Official Source Repository*. https://github.com/espressif/esp-idf. 2024.

[43] *SystemRDL for System-on-Chip Register Description*. 2018.

[44] Keutzer, K., Malik, S., and Newton, A. R. *Orthogonalization of Design Concerns in System-Level Design*. Kluwer Academic Publishers, 2005.

[45] Herdt, V., Große, D., and Drechsler, R. *Virtual Prototyping: Principles, Techniques, and Perspectives*. Springer, 2022. DOI: 10.1007/978-3-030-94234-3.

[46] Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Hammond, I., Strub, F., Laptev, I., and Sivic, J. Flamingo: A Visual Language Model for Few-Shot Learning. In: *arXiv preprint arXiv:2204.14198*, 2022. Advances in Neural Information Processing Systems (NeurIPS).

[47] Ghinami, M., Sehm, R., Hübner, C., Meyer, R., and Berekovic, M. Automated Synthesis of SystemC Transactional Models from Hardware Specifications. In: *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE/ACM. 2025, pp. 1–7.